

# IAR Embedded Workbench®

## C-SPY® Debugging Guide

for Atmel® Corporation's  
**AVR Microcontroller Family**



## **COPYRIGHT NOTICE**

© 2011–2015 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**

IAR Systems, IAR Embedded Workbench, C-SPY, C-RUN, C-STAT, visualSTATE, Focus on Your Code, IAR KickStart Kit, IAR Experiment!, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Atmel and AVR are registered trademarks of Atmel® Corporation.

Adobe and Acrobat Reader are registered trademarks of Adobe Systems Incorporated.

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

## **EDITION NOTICE**

Fourth edition: May 2015

Part number: UCSAVR-4

This guide applies to version 6.x of IAR Embedded Workbench® for Atmel® Corporation's AVR microcontroller family.

Internal reference: M18, Hom7.2, IMAE.

# Brief contents

Tables .....	19
Preface .....	21
<b>Part 1. Basic debugging</b> .....	<b>27</b>
The IAR C-SPY Debugger .....	29
Getting started using C-SPY .....	51
Executing your application .....	69
Variables and expressions .....	87
Breakpoints .....	113
Memory and registers .....	141
<b>Part 2. Analyzing your application</b> .....	<b>163</b>
Trace .....	165
Profiling .....	183
Code coverage .....	193
<b>Part 3. Advanced debugging</b> .....	<b>197</b>
Interrupts .....	199
C-SPY macros .....	207
The C-SPY command line utility— <code>cspybat</code> .....	265
<b>Part 4. Additional reference information</b> .....	<b>291</b>
Debugger options .....	293
Additional information on C-SPY drivers .....	329

Index ..... 347

# Contents

Tables .....	19
Preface .....	21
<b>Who should read this guide</b> .....	21
Required knowledge .....	21
<b>What this guide contains</b> .....	21
Part 1. Basic debugging .....	21
Part 2. Analyzing your application .....	22
Part 3. Advanced debugging .....	22
Part 4. Additional reference information .....	22
<b>Other documentation</b> .....	22
User and reference guides .....	23
The online help system .....	24
Web sites .....	24
<b>Document conventions</b> .....	24
Typographic conventions .....	25
Naming conventions .....	25
<b>Part I. Basic debugging</b> .....	27
The IAR C-SPY Debugger .....	29
<b>Introduction to C-SPY</b> .....	29
An integrated environment .....	29
General C-SPY debugger features .....	30
RTOS awareness .....	31
<b>Debugger concepts</b> .....	32
C-SPY and target systems .....	32
The debugger .....	33
The target system .....	33
The application .....	33
C-SPY debugger systems .....	33
The ROM-monitor program .....	34

Third-party debuggers .....	34
C-SPY plugin modules .....	34
<b>C-SPY drivers overview .....</b>	<b>35</b>
Differences between the C-SPY drivers .....	35
<b>The IAR C-SPY Simulator .....</b>	<b>36</b>
<b>The C-SPY Atmel-ICE driver .....</b>	<b>37</b>
Features .....	37
Communication overview .....	38
Hardware installation .....	38
<b>The C-SPY JTAGICE3 driver .....</b>	<b>39</b>
Features .....	39
Communication overview .....	40
Hardware installation .....	40
<b>The C-SPY JTAGICE mkII/Dragon driver .....</b>	<b>41</b>
Features .....	41
Communication overview .....	42
Hardware installation .....	42
<b>The C-SPY AVR ONE! driver .....</b>	<b>43</b>
Features .....	43
Communication overview .....	44
Hardware installation .....	44
<b>Legacy C-SPY drivers .....</b>	<b>45</b>
The C-SPY JTAGICE driver .....	45
The C-SPY ICE200 driver .....	47
The C-SPY Crypto Controller ROM-monitor driver .....	48
<b>Getting started using C-SPY .....</b>	<b>51</b>
<b>Setting up C-SPY .....</b>	<b>51</b>
Setting up for debugging .....	51
Executing from reset .....	52
Using a setup macro file .....	52
Selecting a device description file .....	52
Loading plugin modules .....	53

<b>Starting C-SPY</b> .....	53
Starting a debug session .....	53
Loading executable files built outside of the IDE .....	54
Starting a debug session with source files missing .....	54
Loading multiple images .....	55
<b>Adapting for target hardware</b> .....	56
Modifying a device description file .....	56
Initializing target hardware before C-SPY starts .....	56
<b>Running example projects</b> .....	57
Running an example project .....	57
<b>Reference information on starting C-SPY</b> .....	59
C-SPY Debugger main window .....	59
Images window .....	64
Get Alternative File dialog box .....	65
Get Example Projects dialog box .....	66
<b>Executing your application</b> .....	69
<b>Introduction to application execution</b> .....	69
Briefly about application execution .....	69
Source and disassembly mode debugging .....	69
Single stepping .....	70
Stepping speed .....	72
Running the application .....	73
Highlighting .....	74
Call stack information .....	74
Terminal input and output .....	75
Debug logging .....	75
<b>Reference information on application execution</b> .....	76
Disassembly window .....	76
Call Stack window .....	80
Terminal I/O window .....	82
Terminal I/O Log File dialog box .....	83
Debug Log window .....	84
Log File dialog box .....	85

Report Assert dialog box .....	86
Autostep settings dialog box .....	86
<b>Variables and expressions .....</b>	<b>87</b>
<b>Introduction to working with variables and expressions .....</b>	<b>87</b>
Briefly about working with variables and expressions .....	87
C-SPY expressions .....	88
Limitations on variable information .....	90
<b>Working with variables and expressions .....</b>	<b>91</b>
Using the windows related to variables and expressions .....	91
Viewing assembler variables .....	91
Getting started using data logging .....	92
<b>Reference information on working with variables and expressions .....</b>	<b>93</b>
Auto window .....	94
Locals window .....	95
Watch window .....	97
Statics window .....	99
Quick Watch window .....	102
Symbols window .....	104
Resolve Symbol Ambiguity dialog box .....	106
Data Log window .....	107
Data Log Summary window .....	109
<b>Breakpoints .....</b>	<b>113</b>
<b>Introduction to setting and using breakpoints .....</b>	<b>113</b>
Reasons for using breakpoints .....	113
Briefly about setting breakpoints .....	113
Breakpoint types .....	114
Breakpoint icons .....	116
Breakpoints in the C-SPY simulator .....	116
Breakpoints in the C-SPY hardware debugger drivers .....	116
Breakpoint consumers .....	118
<b>Setting breakpoints .....</b>	<b>119</b>
Various ways to set a breakpoint .....	119



Toggling a simple code breakpoint .....	120
Setting breakpoints using the dialog box .....	120
Setting a data breakpoint in the Memory window .....	121
Setting breakpoints using system macros .....	122
Useful breakpoint hints .....	123
<b>Reference information on breakpoints .....</b>	<b>124</b>
Breakpoints window .....	125
Breakpoint Usage window .....	126
Code breakpoints dialog box .....	127
Log breakpoints dialog box .....	129
Data breakpoints dialog box .....	130
Data Log breakpoints dialog box .....	132
Immediate breakpoints dialog box .....	133
Complex breakpoints dialog box .....	135
Enter Location dialog box .....	138
Resolve Source Ambiguity dialog box .....	139
<b>Memory and registers .....</b>	<b>141</b>
<b>Introduction to monitoring memory and registers .....</b>	<b>141</b>
Briefly about monitoring memory and registers .....	141
C-SPY memory zones .....	142
Stack display .....	143
<b>Monitoring memory and registers .....</b>	<b>144</b>
Defining application-specific register groups .....	144
<b>Reference information on memory and registers .....</b>	<b>145</b>
Memory window .....	146
Memory Save dialog box .....	150
Memory Restore dialog box .....	151
Fill dialog box .....	152
Symbolic Memory window .....	153
Stack window .....	156
Register window .....	159

<b>Part 2. Analyzing your application</b> .....	163
<b>Trace</b> .....	165
<b>Introduction to using trace</b> .....	165
Reasons for using trace .....	165
Briefly about trace .....	165
Requirements for using trace .....	166
<b>Collecting and using trace data</b> .....	166
Getting started with trace .....	166
Trace data collection using breakpoints .....	166
Searching in trace data .....	167
Browsing through trace data .....	167
<b>Reference information on trace</b> .....	168
Trace window .....	168
Function Trace window .....	170
Timeline window .....	171
Viewing Range dialog box .....	177
Trace Start breakpoints dialog box .....	178
Trace Stop breakpoints dialog box .....	179
Trace Expressions window .....	180
Find in Trace dialog box .....	181
Find in Trace window .....	182
<b>Profiling</b> .....	183
<b>Introduction to the profiler</b> .....	183
Reasons for using the profiler .....	183
Briefly about the profiler .....	183
Requirements for using the profiler .....	184
<b>Using the profiler</b> .....	184
Getting started using the profiler on function level .....	184
Analyzing the profiling data .....	185
Getting started using the profiler on instruction level .....	187
<b>Reference information on the profiler</b> .....	188
Function Profiler window .....	188

Code coverage .....	193
<b>Introduction to code coverage</b> .....	193
Reasons for using code coverage .....	193
Briefly about code coverage .....	193
Requirements and restrictions for using code coverage .....	193
<b>Reference information on code coverage</b> .....	193
Code Coverage window .....	194
<b>Part 3. Advanced debugging</b> .....	197
Interrupts .....	199
<b>Introduction to interrupts</b> .....	199
Briefly about the interrupt simulation system .....	199
Interrupt characteristics .....	200
C-SPY system macros for interrupt simulation .....	201
Target-adapting the interrupt simulation system .....	201
<b>Using the interrupt system</b> .....	202
Simulating a simple interrupt .....	202
<b>Reference information on interrupts</b> .....	203
Interrupts dialog box .....	204
C-SPY macros .....	207
<b>Introduction to C-SPY macros</b> .....	207
Reasons for using C-SPY macros .....	207
Briefly about using C-SPY macros .....	208
Briefly about setup macro functions and files .....	208
Briefly about the macro language .....	208
<b>Using C-SPY macros</b> .....	209
Registering C-SPY macros—an overview .....	210
Executing C-SPY macros—an overview .....	210
Registering and executing using setup macros and setup files .....	211
Executing macros using Quick Watch .....	211
Executing a macro by connecting it to a breakpoint .....	212
Aborting a C-SPY macro .....	213

<b>Reference information on the macro language</b> .....	214
Macro functions .....	214
Macro variables .....	214
Macro parameters .....	215
Macro strings .....	215
Macro statements .....	216
Formatted output .....	217
<b>Reference information on reserved setup macro function names</b> .....	219
execUserPreload .....	219
execUserExecutionStarted .....	220
execUserExecutionStopped .....	220
execUserSetup .....	220
execUserPreReset .....	221
execUserReset .....	221
execUserExit .....	221
<b>Reference information on C-SPY system macros</b> .....	221
__cancelAllInterrupts .....	223
__cancelInterrupt .....	224
__clearBreak .....	224
__closeFile .....	225
__delay .....	225
__disableInterrupts .....	225
__driverType .....	226
__enableInterrupts .....	226
__evaluate .....	227
__fillMemory8 .....	227
__fillMemory16 .....	228
__fillMemory32 .....	229
__getCycleCounter .....	230
__isBatchMode .....	230
__loadImage .....	231
__memoryRestore .....	232
__memoryRestoreFromFile .....	233

__memorySave .....	233
__memorySaveToFile .....	234
__messageBoxYesCancel .....	235
__messageBoxYesNo .....	235
__openFile .....	236
__orderInterrupt .....	237
__readFile .....	238
__readFileByte .....	239
__readMemory8, __readMemoryByte .....	239
__readMemory16 .....	240
__readMemory32 .....	240
__registerMacroFile .....	241
__resetFile .....	241
__setCodeBreak .....	242
__setComplexBreak .....	243
__setDataBreak .....	245
__setLogBreak .....	247
__setSimBreak .....	248
__setTraceStartBreak .....	249
__setTraceStopBreak .....	250
__sourcePosition .....	251
__strFind .....	251
__subString .....	252
__targetDebuggerVersion .....	252
__toLower .....	253
__toString .....	253
__toUpper .....	254
__transparent .....	254
__unloadImage .....	255
__writeFile .....	255
__writeFileByte .....	256
__writeMemory8, __writeMemoryByte .....	256
__writeMemory16 .....	257
__writeMemory32 .....	257

<b>Graphical environment for macros</b> .....	258
Macro Registration window .....	258
Debugger Macros window .....	260
Macro Quicklaunch window .....	262
<b>The C-SPY command line utility—cspybat</b> .....	265
<b>Using C-SPY in batch mode</b> .....	265
Starting cspybat .....	265
Output .....	266
Invocation syntax .....	266
<b>Summary of C-SPY command line options</b> .....	267
General cspybat options .....	267
Options available for all C-SPY drivers .....	268
Options available for the simulator driver .....	269
Options available for all C-SPY hardware debugger drivers .....	269
Options available for the C-SPY Atmel-ICE driver .....	269
Options available for the C-SPY JTAGICE driver, the C-SPY JTAGICE mkII driver, the C-SPY Dragon driver, the C-SPY JTAGICE driver, and the C-SPY AVR ONE! driver .....	269
Options available for the C-SPY JTAGICE mkII driver, the C-SPY Dragon driver, the C-SPY Atmel-ICE driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver .....	270
Options available for the C-SPY JTAGICE driver, the C-SPY JTAGICE mkII driver, and the C-SPY Dragon driver .....	270
Options available for the C-SPY Atmel-ICE driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver .....	270
Options available for the C-SPY JTAGICE mkII driver and the C-SPY Dragon driver .....	270
Options available for the C-SPY Dragon driver .....	271
Options available for the C-SPY ICE200 driver .....	271
<b>Reference information on C-SPY command line options</b> ...	271
--64bit_doubles .....	271
--64k_flash .....	271
--avrone_jtag_clock .....	272

--backend .....	272
--code_coverage_file .....	272
--cpu .....	273
--cycles .....	273
-d .....	274
--debugfile .....	274
--disable_internal_eeprom .....	275
--disable_interrupts .....	275
--download_only .....	275
--drv_atmel_ice .....	276
--drv_communication .....	276
--drv_communication_log .....	276
--drv_debug_port .....	277
--drv_download_data .....	277
--drv_dragon .....	278
--drv_preserve_app_section .....	278
--drv_preserve_boot_section .....	278
--drv_set_exit_breakpoint .....	279
--drv_set_getchar_breakpoint .....	279
--drv_set_putchar_breakpoint .....	280
--drv_suppress_download .....	280
--drv_use_PDI .....	281
--drv_verify_download .....	281
--eeprom_size .....	281
--enhanced_core .....	282
-f .....	282
--ice200_restore_EEPROM .....	282
--ice200_single_step_timers .....	283
--jtagice_clock .....	283
--jtagice_do_hardware_reset .....	283
--jtagice_leave_timers_running .....	284
--jtagice_preserve_eeprom .....	284
--jtagice_restore_fuse .....	285
--jtagicemkII_use_software_breakpoints .....	285

--leave_running .....	285
--macro .....	286
--macro_param .....	286
-p .....	287
--plugin .....	287
--silent .....	288
--timeout .....	288
-v .....	288

## **Part 4. Additional reference information** ..... 291

### **Debugger options** ..... 293

#### **Setting debugger options** ..... 293

#### **Reference information on debugger options** ..... 294

Setup ..... 295

Images ..... 296

Plugins ..... 297

#### **Reference information on C-SPY hardware debugger driver options** ..... 298

AVR ONE! 1 ..... 298

AVR ONE! 2 ..... 301

Communication ..... 302

Extra Options ..... 303

CCR ..... 304

Serial Port ..... 305

ICE200 ..... 307

JTAGICE 1 ..... 309

JTAGICE 2 ..... 311

Atmel-ICE 1 ..... 312

Atmel-ICE 2 ..... 314

JTAGICE3 1 ..... 316

JTAGICE3 2 ..... 318

JTAGICE mkII 1 ..... 319

JTAGICE mkII 2 ..... 322



Dragon 1 .....	323
Dragon 2 .....	325
Third-Party Driver options .....	326
<b>Additional information on C-SPY drivers .....</b>	<b>329</b>
<b>Reference information on C-SPY driver menus .....</b>	<b>329</b>
<i>C-SPY driver</i> .....	329
Simulator menu .....	330
JTAGICE menu .....	331
JTAGICE mkII menu .....	331
Dragon menu .....	332
Atmel-ICE menu .....	332
JTAGICE3 menu .....	333
AVR ONE! menu .....	333
ICE200 menu .....	334
CCR menu .....	334
<b>Reference information on the C-SPY hardware debugger</b>	
<b>drivers .....</b>	<b>335</b>
Fuse Handler dialog box .....	335
Fuse Handler dialog box .....	338
ICE200 Options dialog box .....	340
<b>Resolving problems .....</b>	<b>342</b>
No contact with the target hardware .....	343
Monitor works, but application will not run .....	343
Optimizing downloads .....	343
Using C-SPY macros for transparent commands .....	343
Slow stepping speed .....	344
<b>Index .....</b>	<b>347</b>



# Tables

1: Typographic conventions used in this guide .....	25
2: Naming conventions used in this guide .....	25
3: Driver differences .....	35
4: C-SPY assembler symbols expressions .....	89
5: Handling name conflicts between hardware registers and assembler labels .....	89
6: Available breakpoints in C-SPY hardware debugger drivers .....	117
7: C-SPY macros for breakpoints .....	122
8: Supported graphs in the Timeline window .....	172
9: Project options for enabling the profiler .....	184
10: Project options for enabling code coverage .....	194
11: Timer interrupt settings .....	203
12: Examples of C-SPY macro variables .....	215
13: Summary of system macros .....	221
14: __cancelInterrupt return values .....	224
15: __disableInterrupts return values .....	225
16: __driverType return values .....	226
17: __enableInterrupts return values .....	226
18: __evaluate return values .....	227
19: __isBatchMode return values .....	231
20: __loadImage return values .....	231
21: __messageBoxYesCancel return values .....	235
22: __messageBoxYesNo return values .....	235
23: __openFile return values .....	236
24: __readFile return values .....	238
25: __setCodeBreak return values .....	242
26: __set Complex Break return values .....	245
27: __setDataBreak return values .....	246
28: __setLogBreak return values .....	247
29: __setSimBreak return values .....	248
30: __setTraceStartBreak return values .....	249
31: __setTraceStopBreak return values .....	250

32: __sourcePosition return values .....	251
33: __unloadImage return values .....	255
34: cspybat parameters .....	266
35: Options specific to the C-SPY drivers you are using .....	293

# Preface

Welcome to the *C-SPY® Debugging Guide*. The purpose of this guide is to help you fully use the features in the IAR C-SPY® Debugger for debugging your application based on the AVR microcontroller.

---

## Who should read this guide

Read this guide if you plan to develop an application using IAR Embedded Workbench and want to get the most out of the features available in C-SPY.

### REQUIRED KNOWLEDGE

To use the tools in IAR Embedded Workbench, you should have working knowledge of:

- The architecture and instruction set of the Atmel AVR microcontroller (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation*, page 22.

---

## What this guide contains

Below is a brief outline and summary of the chapters in this guide.

**Note:** Some of the screenshots in this guide are taken from a similar product and not from IAR Embedded Workbench for AVR.

### PART I. BASIC DEBUGGING

- *The IAR C-SPY Debugger* introduces you to the C-SPY debugger and to the concepts that are related to debugging in general and to C-SPY in particular. The chapter also introduces the various C-SPY drivers. The chapter briefly shows the difference in functionality that the various C-SPY drivers provide.
- *Getting started using C-SPY* helps you get started using C-SPY, which includes setting up, starting, and adapting C-SPY for target hardware.

- *Executing your application* describes the conceptual differences between source and disassembly mode debugging, the facilities for executing your application, and finally, how you can handle terminal input and output.
- *Variables and expressions* describes the syntax of the expressions and variables used in C-SPY, as well as the limitations on variable information. The chapter also demonstrates the various methods for monitoring variables and expressions.
- *Breakpoints* describes the breakpoint system and the various ways to set breakpoints.
- *Memory and registers* shows how you can examine memory and registers.

## **PART 2. ANALYZING YOUR APPLICATION**

- *Trace* describes how you can inspect the program flow up to a specific state using trace data.
- *Profiling* describes how the profiler can help you find the functions in your application source code where the most time is spent during execution.
- *Code coverage* describes how the code coverage functionality can help you verify whether all parts of your code have been executed, thus identifying parts which have not been executed.

## **PART 3. ADVANCED DEBUGGING**

- *Interrupts* contains detailed information about the C-SPY interrupt simulation system and how to configure the simulated interrupts to make them reflect the interrupts of your target hardware.
- *C-SPY macros* describes the C-SPY macro system, its features, the purposes of these features, and how to use them.
- *The C-SPY command line utility—`cspybat`* describes how to use C-SPY in batch mode.

## **PART 4. ADDITIONAL REFERENCE INFORMATION**

- *Debugger options* describes the options you must set before you start the C-SPY debugger.
- *Additional information on C-SPY drivers* describes menus and features provided by the C-SPY drivers not described in any dedicated topics.

---

## **Other documentation**

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information

Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

## USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, is available in the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide*.
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for AVR*.
- Programming for the IAR C/C++ Compiler for AVR, is available in the *IAR C/C++ Compiler Reference Guide for AVR*.
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, is available in the *IAR Linker and Library Tools Reference Guide*.
- Programming for the IAR Assembler for AVR, is available in the *AVR® IAR Assembler Reference Guide*.
- Using the IAR DLIB Library, is available in the *DLIB Library Reference information*, available in the online help system.
- Using the IAR CLIB Library, is available in the *IAR C Library Functions Reference Guide*, available in the online help system.
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT Static Analysis Guide*.
- Developing safety-critical applications using the MISRA C guidelines, is available in the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for AVR, is available in the *IAR Embedded Workbench® Migration Guide*.

**Note:** Additional documentation might be available depending on your product installation.

## THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about project management, editing, and building in the IDE
- Information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1. Note that if you select a function name in the editor window and press F1 while using the CLIB library, you will get reference information for the DLIB library.

## WEB SITES

Recommended web sites:

- The Atmel® Corporation web site, [www.atmel.com](http://www.atmel.com), that contains information and news about the Atmel AVR microcontrollers.
- The IAR Systems web site, [www.iar.com](http://www.iar.com), that holds application notes and other product information.
- The web site of the C standardization working group, [www.open-std.org/jtc1/sc22/wg14](http://www.open-std.org/jtc1/sc22/wg14).
- The web site of the C++ Standards Committee, [www.open-std.org/jtc1/sc22/wg21](http://www.open-std.org/jtc1/sc22/wg21).
- Finally, the Embedded C++ Technical Committee web site, [www.caravan.net/ec2plus](http://www.caravan.net/ec2plus), that contains information about the Embedded C++ standard.

---

## Document conventions

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

When referring to a directory in your product installation, for example `avr\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 7.n\avr\doc`.



## TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:





Style	Used for
computer	<ul style="list-style-type: none"> <li>• Source code examples and file paths.</li> <li>• Text on the command line.</li> <li>• Binary, hexadecimal, and octal numbers.</li> </ul>
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a directive, where [ and ] are not part of the actual directive, but any [, ], {, or } are part of the directive syntax.
{option}	A mandatory part of a directive, where { and } are not part of the actual directive, but any [, ], {, or } are part of the directive syntax.
[option]	An optional part of a command.
[a b c]	An optional part of a command with alternatives.
{a b c}	A mandatory part of a command with alternatives.
<b>bold</b>	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> <li>• A cross-reference within this guide or to another guide.</li> <li>• Emphasis.</li> </ul>
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

## NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for AVR	IAR Embedded Workbench®

Table 2: Naming conventions used in this guide

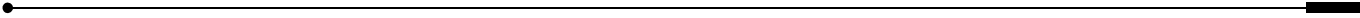
<b>Brand name</b>	<b>Generic term</b>
IAR Embedded Workbench® IDE for AVR	the IDE
IAR C-SPY® Debugger for AVR	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for AVR	the compiler
IAR Assembler™ for AVR	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Library™	the DLIB library
IAR CLIB Library™	the CLIB library

*Table 2: Naming conventions used in this guide (Continued)*

# Part I. Basic debugging

This part of the *C-SPY® Debugging Guide for AVR* includes these chapters:

- The IAR C-SPY Debugger
- Getting started using C-SPY
- Executing your application
- Variables and expressions
- Breakpoints
- Memory and registers





# The IAR C-SPY Debugger

- Introduction to C-SPY
- Debugger concepts
- C-SPY drivers overview
- The IAR C-SPY Simulator
- The C-SPY Atmel-ICE driver
- The C-SPY JTAGICE3 driver
- The C-SPY JTAGICE mkII/Dragon driver
- The C-SPY AVR ONE! driver
- Legacy C-SPY drivers

---

## Introduction to C-SPY

These topics are covered:

- An integrated environment
- General C-SPY debugger features
- RTOS awareness

### **AN INTEGRATED ENVIRONMENT**

C-SPY is a high-level-language debugger for embedded applications. It is designed for use with the IAR Systems compilers and assemblers, and is completely integrated in the IDE, providing development and debugging within the same application. This will give you possibilities such as:

- Editing while debugging. During a debug session, you can make corrections directly in the same source code window that is used for controlling the debugging. Changes will be included in the next project rebuild.
- Setting breakpoints at any point during the development cycle. You can inspect and modify breakpoint definitions also when the debugger is not running, and breakpoint definitions flow with the text as you edit. Your debug settings, such as

watch properties, window layouts, and register groups will be preserved between your debug sessions.

All windows that are open in the Embedded Workbench workspace will stay open when you start the C-SPY Debugger. In addition, a set of C-SPY-specific windows are opened.

## GENERAL C-SPY DEBUGGER FEATURES

Because IAR Systems provides an entire toolchain, the output from the compiler and linker can include extensive debug information for the debugger, resulting in good debugging possibilities for you.

C-SPY offers these general features:

- Source and disassembly level debugging
 

C-SPY allows you to switch between source and disassembly debugging as required, for both C or C++ and assembler source code.
- Single-stepping on a function call level
 

Compared to traditional debuggers, where the finest granularity for source level stepping is line by line, C-SPY provides a finer level of control by identifying every statement and function call as a step point. This means that each function call—inside expressions, and function calls that are part of parameter lists to other functions—can be single-stepped. The latter is especially useful when debugging C++ code, where numerous extra function calls are made, for example to object constructors.
- Code and data breakpoints
 

The C-SPY breakpoint system lets you set breakpoints of various kinds in the application being debugged, allowing you to stop at locations of particular interest. For example, you set breakpoints to investigate whether your program logic is correct or to investigate how and when the data changes.
- Monitoring variables and expressions
 

For variables and expressions there is a wide choice of facilities. You can easily monitor values of a specified set of variables and expressions, continuously or on demand. You can also choose to monitor only local variables, static variables, etc.
- Container awareness
 

When you run your application in C-SPY, you can view the elements of library data types such as STL lists and vectors. This gives you a very good overview and debugging opportunities when you work with C++ STL containers.
- Call stack information
 

The compiler generates extensive call stack information. This allows the debugger to show, without any runtime penalty, the complete stack of function calls wherever the

program counter is. You can select any function in the call stack, and for each function you get valid information for local variables and available registers.

- Powerful macro system

C-SPY includes a powerful internal macro system, to allow you to define complex sets of actions to be performed. C-SPY macros can be used on their own or in conjunction with complex breakpoints and—if you are using the simulator—the interrupt simulation system to perform a wide variety of tasks.

### **Additional general C-SPY debugger features**

This list shows some additional features:

- Threaded execution keeps the IDE responsive while running the target application
- Automatic stepping
- The source browser provides easy navigation to functions, types, and variables
- Extensive type recognition of variables
- Configurable registers (CPU and peripherals) and memory windows
- Graphical stack view with overflow detection
- Support for code coverage and function level profiling
- The target application can access files on the host PC using file I/O (requires the DLIB library)
- UBROF, Intel-extended, and Motorola input formats supported
- Optional terminal I/O emulation.

### **RTOS AWARENESS**

C-SPY supports RTOS-aware debugging.

These operating systems are currently supported:

- Micrium uC/OS-II
- OSEK Run Time Interface (ORTI)

RTOS plugin modules can be provided by IAR Systems, and by third-party suppliers. Contact your software distributor or IAR Systems representative, alternatively visit the IAR Systems web site, for information about supported RTOS modules.

A C-SPY RTOS awareness plugin module gives you a high level of control and visibility over an application built on top of an RTOS. It displays RTOS-specific items like task lists, queues, semaphores, mailboxes, and various RTOS system variables. Task-specific breakpoints and task-specific stepping make it easier to debug tasks.

A loaded plugin will add its own menu, set of windows, and buttons when a debug session is started (provided that the RTOS is linked with the application). For

information about other RTOS awareness plugin modules, refer to the manufacturer of the plugin module.

---

## Debugger concepts

This section introduces some of the concepts and terms that are related to debugging in general and to C-SPY in particular. This section does not contain specific information related to C-SPY features. Instead, you will find such information in the other chapters of this documentation. The IAR Systems user documentation uses the terms described in this section when referring to these concepts.

These topics are covered:

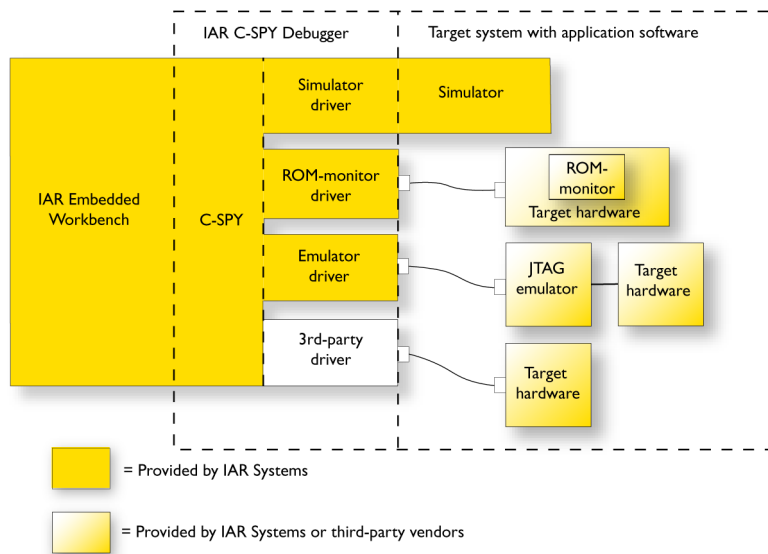
- C-SPY and target systems
- The debugger
- The target system
- The application
- C-SPY debugger systems
- The ROM-monitor program
- Third-party debuggers
- C-SPY plugin modules

### **C-SPY AND TARGET SYSTEMS**

You can use C-SPY to debug either a software target system or a hardware target system.



This figure gives an overview of C-SPY and possible target systems:



## THE DEBUGGER

The debugger, for instance C-SPY, is the program that you use for debugging your applications on a target system.

## THE TARGET SYSTEM

The target system is the system on which you execute your application when you are debugging it. The target system can consist of hardware, either an evaluation board or your own hardware design. It can also be completely or partially simulated by software. Each type of target system needs a dedicated C-SPY driver.

## THE APPLICATION

A user application is the software you have developed and which you want to debug using C-SPY.

## C-SPY DEBUGGER SYSTEMS

C-SPY consists of both a general part which provides a basic set of debugger features, and a target-specific back end. The back end consists of two components: a processor module—one for every microcontroller, which defines the properties of the microcontroller, and a *C-SPY driver*. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides the user

interface—menus, windows, and dialog boxes—to the functions provided by the target system, for instance, special breakpoints. Typically, there are three main types of C-SPY drivers:

- Simulator driver
- ROM-monitor driver
- Emulator driver.

C-SPY is available with a simulator driver, and depending on your product package, optional drivers for hardware debugger systems. For an overview of the available C-SPY drivers and the functionality provided by each driver, see *C-SPY drivers overview*, page 35.

## THE ROM-MONITOR PROGRAM

The ROM-monitor program is a piece of firmware that is loaded to non-volatile memory on your target hardware; it runs in parallel with your application. The ROM-monitor communicates with the debugger and provides services needed for debugging the application, for instance stepping and breakpoints.

## THIRD-PARTY DEBUGGERS

You can use a third-party debugger together with the IAR Systems toolchain as long as the third-party debugger can read any of the output formats provided by XLINK, such as UBROF, ELF/DWARF, COFF, Intel-extended, Motorola, or any other available format. For information about which format to use with a third-party debugger, see the user documentation supplied with that tool.

## C-SPY PLUGIN MODULES

C-SPY is designed as a modular architecture with an open SDK that can be used for implementing additional functionality to the debugger in the form of plugin modules. These modules can be seamlessly integrated in the IDE.

Plugin modules are provided by IAR Systems, or can be supplied by third-party vendors. Examples of such modules are:

- Code Coverage, which is integrated in the IDE.
- The various C-SPY drivers for debugging using certain debug systems.
- RTOS plugin modules for support for real-time OS aware debugging.
- C-SPYLink that bridges IAR 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. For more information, see the documentation provided with IAR visualSTATE.

For more information about the C-SPY SDK, contact IAR Systems.

## C-SPY drivers overview

At the time of writing this guide, the IAR C-SPY Debugger for the AVR microcontrollers is available with drivers for these target systems and evaluation boards:

- Simulator
- AVR® Atmel-ICE
- AVR® JTAGICE3
- AVR® JTAGICE mkII/AVR® Dragon
- AVR® AVR ONE!
- AVR® JTAGICE
- AVR® ICE200
- AVR® Crypto Controller ROM-monitor (CCR) for the Atmel Smart Card Development Board (SCDB) and the Voyager development system.

**Note:** In addition to the drivers supplied with IAR Embedded Workbench, you can also load debugger drivers supplied by a third-party vendor; see *Third-Party Driver options*, page 326.

### DIFFERENCES BETWEEN THE C-SPY DRIVERS

This table summarizes the key differences between the C-SPY drivers:

Feature	Simulator	Atmel-ICE	JTAGICE3	JTAGICE mkII/ Dragon	AVR ONE!	JTAGICE	ICE200	CCR
Code breakpoints	x	x <sup>1</sup>	x <sup>1</sup>	x <sup>1</sup>	x <sup>1</sup>	x <sup>1</sup>	x	x
Data breakpoints	x	x	x	x <sup>1</sup>	x	x <sup>1</sup>	--	--
Execution in real time	--	x	x	x	x	x	x	x
Zero memory footprint	x	x	x	x	x	x	x	x
Simulated interrupts	x	--	--	--	--	--	--	--
Real interrupts	--	x	x	x	x	x	x	x

Table 3: Driver differences

Feature	Simulator	Atmel-ICE	JTAGICE3	JTAGICE mkII/ Dragon	AVR ONE!	JTAGICE	ICE200	CCR
Cycle counter	x	--	--	--	--	--	x	--
Code coverage	x	--	--	--	--	--	--	--
Data coverage	x	--	--	--	--	--	--	--
Function/ instruction profiler	x	--	--	--	--	--	--	--
Trace	x	--	--	--	--	--	--	--

Table 3: Driver differences (Continued)

1 With specific requirements or restrictions, see the respective chapter in this guide.

## The IAR C-SPY Simulator

The C-SPY Simulator simulates the functions of the target processor entirely in software, which means that you can debug the program logic long before any hardware is available. Because no hardware is required, it is also the most cost-effective solution for many applications.

The C-SPY Simulator supports:

- Instruction-level simulation
- Memory configuration and validation
- Interrupt simulation
- Peripheral simulation (using the C-SPY macro system in conjunction with immediate breakpoints).

Simulating hardware instead of using a hardware debugging system means that some limitations do not apply, but that there are other limitations instead. For example:

- You can set an unlimited number of breakpoints in the simulator.
- When you stop executing your application, time actually stops in the simulator. When you stop application execution on a hardware debugging system, there might still be activities in the system. For example, peripheral units might still be active and reading from or writing to SFR ports.
- Application execution is significantly much slower in a simulator compared to when using a hardware debugging system. However, during a debug session, this might not necessarily be a problem.
- The simulator is not cycle accurate.

- Peripheral simulation is limited in the C-SPY Simulator and therefore the simulator is suitable mostly for debugging code that does not interact too much with peripheral units.

---

## The C-SPY Atmel-ICE driver

The C-SPY Atmel-ICE driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY Atmel-ICE driver, C-SPY can connect to Atmel-ICE. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

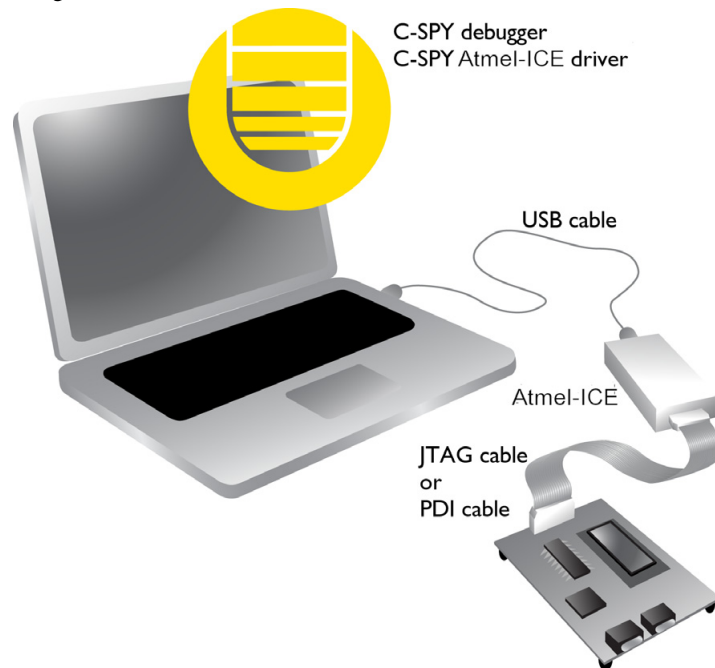
### FEATURES

In addition to the general features of C-SPY, the Atmel-ICE driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via USB
- Fuse handler.

## COMMUNICATION OVERVIEW

The C-SPY Atmel-ICE driver uses the USB port to communicate with Atmel-ICE. Atmel-ICE communicates with the hardware interface—for example JTAG, PDI, debugWIRE, or ISP—on the microcontroller.



When a debugging session is started, your application is automatically downloaded and programmed into flash memory. You can disable this feature, if necessary.

## HARDWARE INSTALLATION

For information about the hardware installation, see the *AVR® Atmel-ICE User Guide* from Atmel® Corporation. This power-up sequence is recommended to ensure proper communication between the target board, Atmel-ICE, and C-SPY:

- 1 Power up the target board.
- 2 Power up Atmel-ICE.
- 3 Start the C-SPY debug session.

To enable the hardware interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded Workbench IDE or a similar tool capable of programming the fuses to check and

program these bits. For more information, see the *Fuse Handler dialog box*, page 335.

---

## The C-SPY JTAGICE3 driver

The C-SPY JTAGICE3 driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY JTAGICE3 driver, C-SPY can connect to JTAGICE3. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

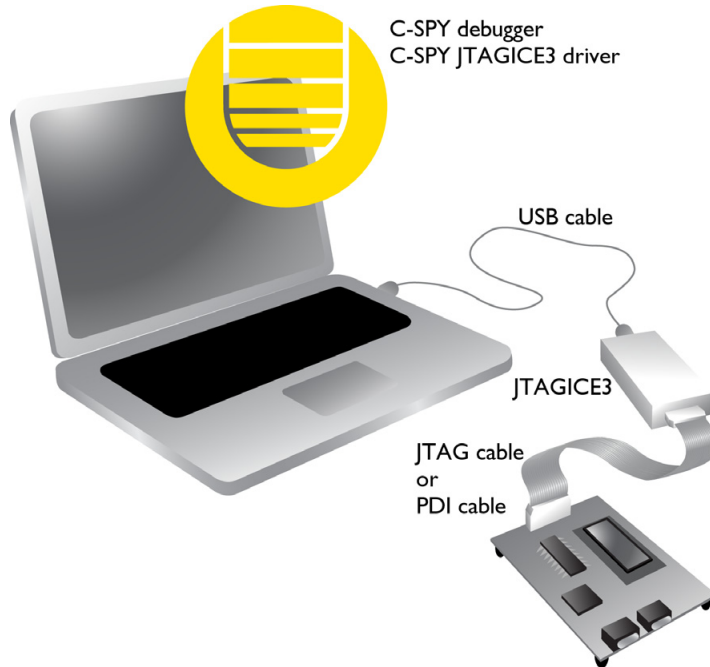
### FEATURES

In addition to the general features of C-SPY, the JTAGICE3 driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via USB
- Fuse handler.

## COMMUNICATION OVERVIEW

The C-SPY JTAGICE3 driver uses the USB port to communicate with Atmel JTAGICE3. JTAGICE3 communicates with the hardware interface—for example JTAG, PDI, debugWIRE, or ISP—on the microcontroller.



When a debugging session is started, your application is automatically downloaded and programmed into flash memory. You can disable this feature, if necessary.

## HARDWARE INSTALLATION

For information about the hardware installation, see the *AVR® JTAGICE3 User Guide* from Atmel® Corporation. This power-up sequence is recommended to ensure proper communication between the target board, JTAGICE3, and C-SPY:

- 1 Power up the target board.
- 2 Power up JTAGICE3.
- 3 Start the C-SPY debug session.

To enable the hardware interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded



Workbench IDE or a similar tool capable of programming the fuses to check and program these bits. For more information, see the *Fuse Handler dialog box*, page 335.

---

## The C-SPY JTAGICE mkII/Dragon driver

The C-SPY JTAGICE mkII driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY JTAGICE mkII driver, C-SPY can connect to JTAGICE mkII and Dragon. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

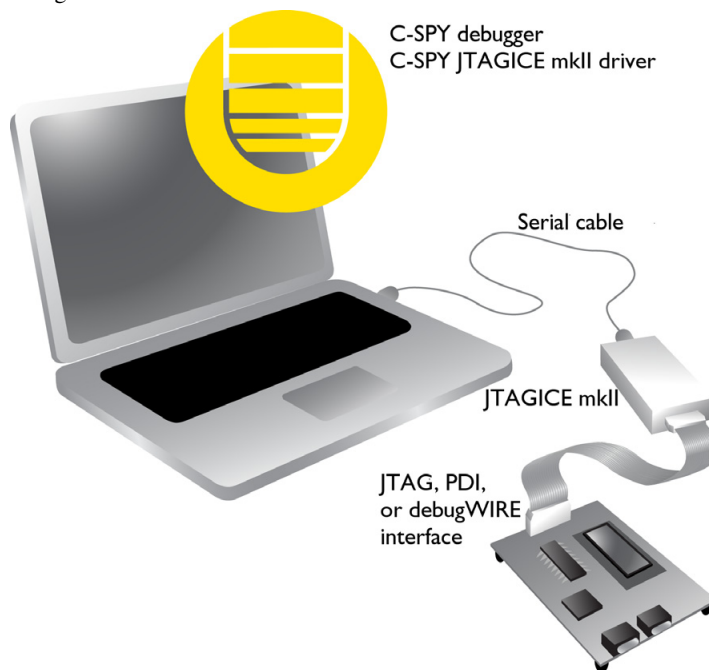
### FEATURES

In addition to the general features of C-SPY, the JTAGICE mkII driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints, for devices that support software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via the serial port or USB
- Fuse handler.

## COMMUNICATION OVERVIEW

The C-SPY JTAGICE mkII driver uses the serial port to communicate with Atmel AVR JTAGICE mkII. JTAGICE mkII communicates with the JTAG, the PDI, or the debugWIRE interface on the microcontroller.



When a debugging session is started, your application is automatically downloaded and programmed into flash memory. You can disable this feature, if necessary.

## HARDWARE INSTALLATION

For information about the hardware installation, see the *AVR® JTAGICE mkII User Guide* from Atmel® Corporation. The following power-up sequence is recommended to ensure proper communication between the target board, JTAGICE mkII, and C-SPY:

- 1 Power up the target board.
- 2 Power up JTAGICE mkII.
- 3 Start the C-SPY debug session.

To enable the JTAG interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded

Workbench IDE or a similar tool capable of programming the fuses to check and program these bits. For more information, see the *Fuse Handler dialog box*, page 335.

---

## The C-SPY AVR ONE! driver

The C-SPY AVR ONE! driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY AVR ONE! driver, C-SPY can connect to AVR ONE!. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

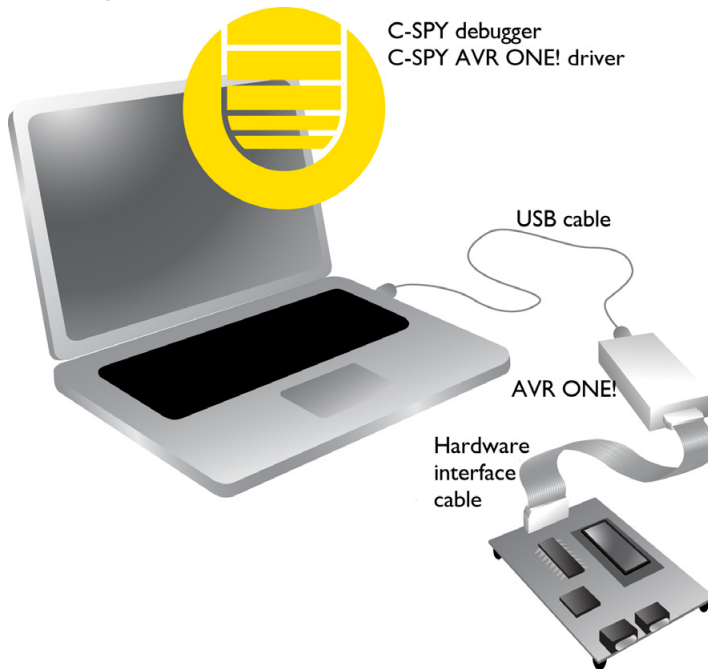
### FEATURES

In addition to the general features of C-SPY, the AVR ONE! driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via USB
- Fuse handler.

## COMMUNICATION OVERVIEW

The C-SPY AVR ONE! driver uses the USB port to communicate with Atmel AVR ONE!. AVR ONE! communicates with the hardware interface—for example JTAG, PDI, debugWIRE, or ISP—on the microcontroller.



When a debugging session is started, your application is automatically downloaded and programmed into flash memory. You can disable this feature, if necessary.

## HARDWARE INSTALLATION

For information about the hardware installation, see the *AVR® AVR ONE! User Guide* from Atmel® Corporation. This power-up sequence is recommended to ensure proper communication between the target board, AVR ONE!, and C-SPY:

- 1 Power up the target board.
- 2 Power up AVR ONE!.
- 3 Start the C-SPY debug session.

To enable the hardware interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded

Workbench IDE or a similar tool capable of programming the fuses to check and program these bits. For more information, see the *Fuse Handler dialog box*, page 338.

---

## Legacy C-SPY drivers

These C-SPY drivers are also available, but are considered legacy:

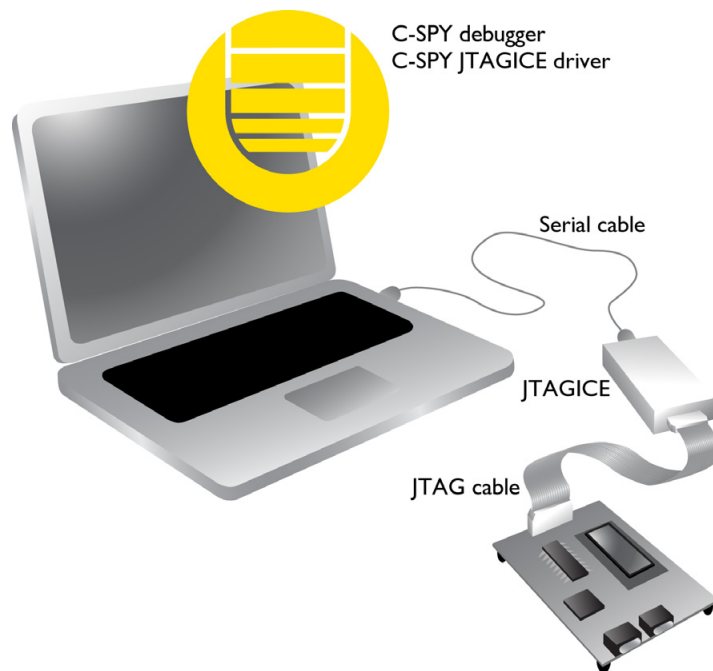
- The C-SPY JTAGICE driver
- The C-SPY ICE200 driver
- The C-SPY Crypto Controller ROM-monitor driver.

### THE C-SPY JTAGICE DRIVER

The C-SPY JTAGICE driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY JTAGICE driver, C-SPY can connect to JTAGICE. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

## Communication overview

The C-SPY JTAGICE driver uses the serial port to communicate with Atmel AVR JTAGICE. JTAGICE communicates with the JTAG interface on the microcontroller.



## Hardware installation

For information about the hardware installation, see the *AVR® JTAGICE User Guide* from Atmel® Corporation. The following power-up sequence is recommended to ensure proper communication between the target board, JTAGICE, and C-SPY:

- 1 Power up the target board.
- 2 Power up JTAGICE.
- 3 Start the C-SPY debug session.

To enable the JTAG interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use a tool capable of programming the fuses to check and program these bits.

## THE C-SPY ICE200 DRIVER

The C-SPY ICE200 driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY ICE200 driver, C-SPY can connect to ICE200, which in turn emulates the AVR microcontroller. Because the hardware debugger logic is built into ICE200, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

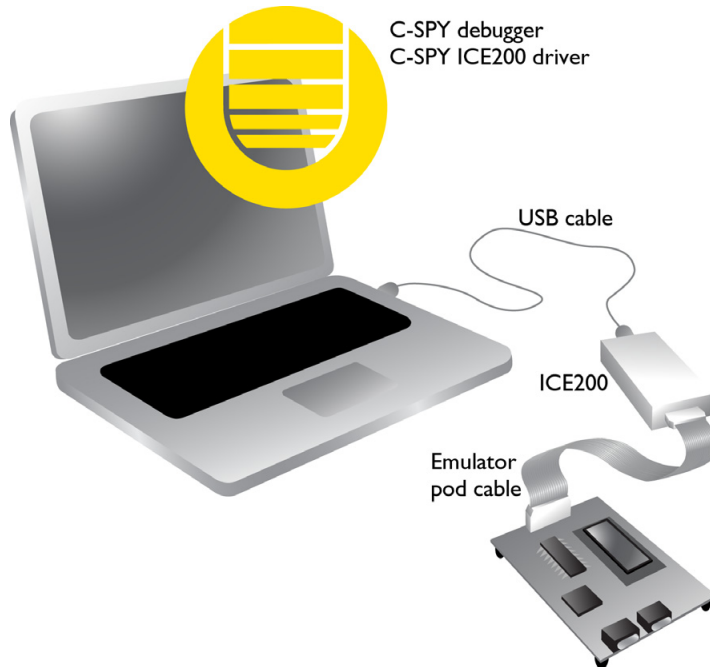
### Features

In addition to the general features of C-SPY, the ICE200 driver also provides:

- Execution in real time with full access to the microcontroller
- Unlimited number of code breakpoints
- Zero memory footprint on the target system.

### Communication overview

The C-SPY ICE200 driver uses the serial port to communicate with the ICE200 emulator, which is connected to the AVR socket on the hardware.



When a debugging session is started, your application is automatically downloaded and programmed into flash memory. You can disable this feature, if necessary.

### **Supported devices**

These devices are supported by the ICE200 debugger system:

AT90S8515, AT90S4414, AT90S8535, AT90S4434, AT90S4433, AT90S2333, AT90S2313, ATtiny12, AT90S2323, AT90S2343

### **Hardware installation**

For information about the hardware installation, see the *AVR® ICE200 User Guide* from Atmel® Corporation. The target hardware must provide both power supply and a clock source. ICE200 will not function unless these conditions are met.

### **THE C-SPY CRYPTO CONTROLLER ROM-MONITOR DRIVER**

The C-SPY Crypto Controller ROM-monitor (CCR) driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY CCR driver, C-SPY can connect to the CCR for the Atmel Smart Card Development Board (SCDB) and the Voyager development system.

### **Features**

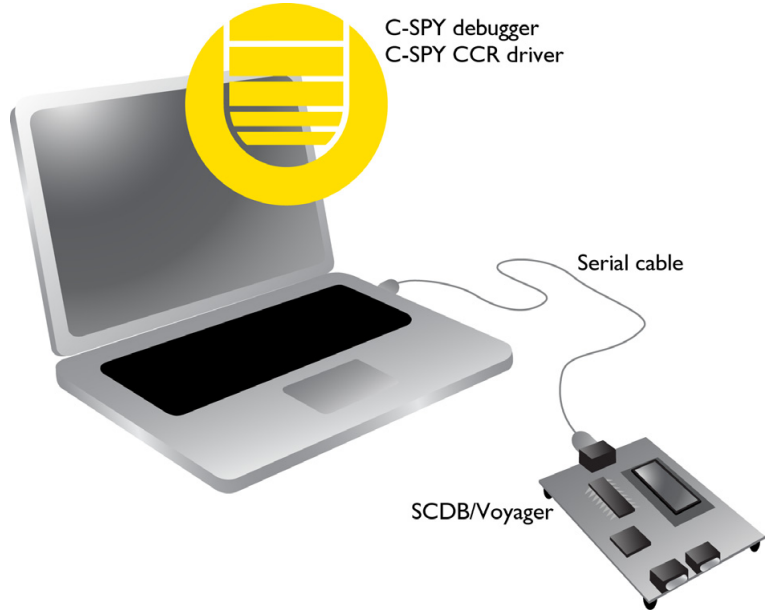
In addition to the general features of C-SPY, the CCR driver also provides:

- Execution in real time
- Communication via serial cable
- Support for real interrupts.



### Communication overview

The C-SPY CCR driver uses the serial port to communicate with the Atmel SCDB/Voyager development system.



When a debugging session is started, your application is automatically downloaded and programmed into flash memory. You can disable this feature, if necessary.



# Getting started using C-SPY

- Setting up C-SPY
- Starting C-SPY
- Adapting for target hardware
- Running example projects
- Reference information on starting C-SPY

---

## Setting up C-SPY

These tasks are covered:

- Setting up for debugging
- Executing from reset
- Using a setup macro file
- Selecting a device description file
- Loading plugin modules

### SETTING UP FOR DEBUGGING

- 1 Before you start C-SPY, choose **Project>Options>Debugger>Setup** and select the C-SPY driver that matches your debugger system: simulator or a hardware debugger system.
- 2 In the **Category** list, select the appropriate C-SPY driver and make your settings.  
For information about these options, see *Debugger options*, page 293.
- 3 Click **OK**.
- 4 Choose **Tools>Options** to open the **IDE Options** dialog box:
  - Select **Debugger** to configure the debugger behavior
  - Select **Stack** to configure the debugger's tracking of stack usage.

For more information about these options, see the *IDE Project Management and Building Guide*.

See also *Adapting for target hardware*, page 56.

## EXECUTING FROM RESET

The **Run to** option—available on the **Debugger>Setup** page—specifies a location you want C-SPY to run to when you start a debug session as well as after each reset. C-SPY will place a temporary breakpoint at this location and all code up to this point is executed before stopping at the location.

The default location to run to is the `main` function. Type the name of the location if you want C-SPY to run to a different location. You can specify assembler labels or whatever can be evaluated to such, for instance function names.

If you leave the check box empty, the program counter will then contain the regular hardware reset address at each reset. Or, if you have selected the option **Get reset address from UBROF**, the program counter will contain the `__program_start` label.

If no breakpoints are available when C-SPY starts, a warning message notifies you that single stepping will be required and that this is time-consuming. You can then continue execution in single-step mode or stop at the first instruction. If you choose to stop at the first instruction, the debugger starts executing with the PC (program counter) at the default reset location instead of the location you typed in the **Run to** box.

**Note:** This message will never be displayed in the C-SPY Simulator, where breakpoints are unlimited.

## USING A SETUP MACRO FILE

A setup macro file is a macro file that you choose to load automatically when C-SPY starts. You can define the setup macro file to perform actions according to your needs, using setup macro functions and system macros. Thus, if you load a setup macro file you can initialize C-SPY to perform actions automatically.

For more information about setup macro files and functions, see *Introduction to C-SPY macros*, page 207. For an example of how to use a setup macro file, see *Initializing target hardware before C-SPY starts*, page 56.

### To register a setup macro file:

- 1 Before you start C-SPY, choose **Project>Options>Debugger>Setup**.
- 2 Select **Use macro file** and type the path and name of your setup macro file, for example `Setup.mac`. If you do not type a filename extension, the extension `mac` is assumed.

## SELECTING A DEVICE DESCRIPTION FILE

C-SPY uses device description files to handle device-specific information.

A default device description file is automatically used based on your project settings. If you want to override the default file, you must select your device description file. Device description files are provided in the `AVR\config` directory and they have the filename extension `ddf`.

For more information about device description files, see *Adapting for target hardware*, page 56.

#### To override the default device description file:

- 1 Before you start C-SPY, choose **Project>Options>Debugger>Setup**.
- 2 Enable the use of a device description file and select a file using the **Device description file** browse button.

**Note:** You can easily view your device description files that are used for your project. Choose **Project>Open Device Description File** and select the file you want to view.

### LOADING PLUGIN MODULES

On the **Plugins** page you can specify C-SPY plugin modules to load and make available during debug sessions. Plugin modules can be provided by IAR Systems, and by third-party suppliers. Contact your software distributor or IAR Systems representative, or visit the IAR Systems web site, for information about available modules.

For more information, see *Plugins*, page 297.

---

## Starting C-SPY

When you have set up the debugger, you are ready to start a debug session.

These tasks are covered:

- Starting a debug session
- Loading executable files built outside of the IDE
- Starting a debug session with source files missing
- Loading multiple images

### STARTING A DEBUG SESSION

You can choose to start a debug session with or without loading the current executable file.



To start C-SPY and download the current executable file, click the **Download and Debug** button. Alternatively, choose **Project>Download and Debug**.



To start C-SPY without downloading the current executable file, click the **Debug without Downloading** button. Alternatively, choose **Project>Debug without Downloading**.

## LOADING EXECUTABLE FILES BUILT OUTSIDE OF THE IDE

You can also load C-SPY with an application that was built outside the IDE, for example applications built on the command line. To load an externally built executable file and to set build options you must first create a project for it in your workspace.

### To create a project for an externally built file:

- 1 Choose **Project>Create New Project**, and specify a project name.
- 2 To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the **Files of type** drop-down list. Locate the executable file.



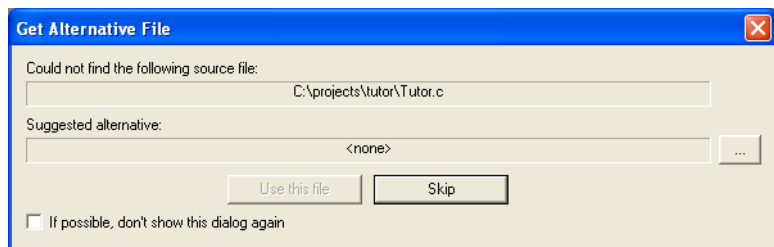
- 3 To start the executable file, click the **Download and Debug** button. The project can be reused whenever you rebuild your executable file.

The only project options that are meaningful to set for this kind of project are options in the **General Options** and **Debugger** categories. Make sure to set up the general project options in the same way as when the executable file was built.

## STARTING A DEBUG SESSION WITH SOURCE FILES MISSING

Normally, when you use the IAR Embedded Workbench IDE to edit source files, build your project, and start the debug session, all required files are available and the process works as expected.

However, if C-SPY cannot automatically find the source files, for example if the application was built on another computer, the **Get Alternative File** dialog box is displayed:



Typically, you can use the dialog box like this:

- The source files are not available: Click **If possible, don't show this dialog again** and then click **Skip**. C-SPY will assume that there simply is no source file available.

The dialog box will not appear again, and the debug session will not try to display the source code.

- Alternative source files are available at another location: Specify an alternative source code file, click **If possible, don't show this dialog again**, and then click **Use this file**. C-SPY will assume that the alternative file should be used. The dialog box will not appear again, unless a file is needed for which there is no alternative file specified and which cannot be located automatically.

If you restart the IAR Embedded Workbench IDE, the **Get Alternative File** dialog box will be displayed again once even if you have clicked **If possible, don't show this dialog again**. This gives you an opportunity to modify your previous settings.

For more information, see *Get Alternative File dialog box*, page 65.

## LOADING MULTIPLE IMAGES

Normally, a debuggable application consists of exactly one file that you debug. However, you can also load additional debug files (images). This means that the complete program consists of several images.

Typically, this is useful if you want to debug your application in combination with a prebuilt ROM image that contains an additional library for some platform-provided features. The ROM image and the application are built using separate projects in the IAR Embedded Workbench IDE and generate separate output files.

If more than one image has been loaded, you will have access to the combined debug information for all the loaded images. In the Images window you can choose whether you want to have access to debug information for one image or for all images.

### To load additional images at C-SPY startup:

- 1 Choose **Project>Options>Debugger>Images** and specify up to three additional images to be loaded. For more information, see *Images*, page 296.
- 2 Start the debug session.

To load additional images at a specific moment:

Use the `__loadImage` system macro and execute it using either one of the methods described in *Using C-SPY macros*, page 209.

To display a list of loaded images:

Choose **Images** from the **View** menu. The **Images** window is displayed, see *Images window*, page 64.

---

## Adapting for target hardware

These topics are covered

- Modifying a device description file
- Initializing target hardware before C-SPY starts

### MODIFYING A DEVICE DESCRIPTION FILE

C-SPY uses device description files provided with the product to handle several of the target-specific adaptations, see *Selecting a device description file*, page 52. They contain device-specific information such as:

- Memory information for device-specific memory zones, see *C-SPY memory zones*, page 142.
- Definitions of memory-mapped peripheral units, device-specific CPU registers, and groups of these.
- Definitions for device-specific interrupts, which makes it possible to simulate these interrupts in the C-SPY simulator; see *Interrupts*, page 199.

Normally, you do not need to modify the device description file. However, if the predefinitions are not sufficient for some reason, you can edit the file. Note, however, that the format of these descriptions might be updated in future upgrades of the product.

Make a copy of the device description file that best suits your needs, and modify it according to the description in the file. Reload the project to make the changes take effect.

For information about how to load a device description file, see *Selecting a device description file*, page 52.

### INITIALIZING TARGET HARDWARE BEFORE C-SPY STARTS

You can use C-SPY macros to initialize target hardware before C-SPY starts. For example, if your hardware uses external memory that must be enabled before code can be downloaded to it, C-SPY needs a macro to perform this action before your application can be downloaded.

- I Create a new text file and define your macro function.

By using the built-in `execUserPreload` setup macro function, your macro function will be executed directly after the communication with the target system is established but before C-SPY downloads your application.



For example, a macro that enables external SDRAM could look like this:

```
/* Your macro function. */
enableExternalSDRAM()
{
    __message "Enabling external SDRAM\n";
    __writeMemory32(...);
}

/* Setup macro determines time of execution. */
execUserPreload()
{
    enableExternalSDRAM();
}
```

- 2 Save the file with the filename extension `mac`.
- 3 Before you start C-SPY, choose **Project>Options>Debugger** and click the **Setup** tab.
- 4 Select the option **Use Setup file** and choose the macro file you just created.

Your setup macro will now be loaded during the C-SPY startup sequence.

---

## Running example projects

These tasks are covered:

- Running an example project

### RUNNING AN EXAMPLE PROJECT

Example applications are provided with IAR Embedded Workbench. You can use these examples to get started using the development tools from IAR Systems. You can also use the examples as a starting point for your application project.

You can find the examples in the `AVR\examples` directory. The examples are ready to be used as is. They are supplied with ready-made workspace files, together with source code files and all other related files.

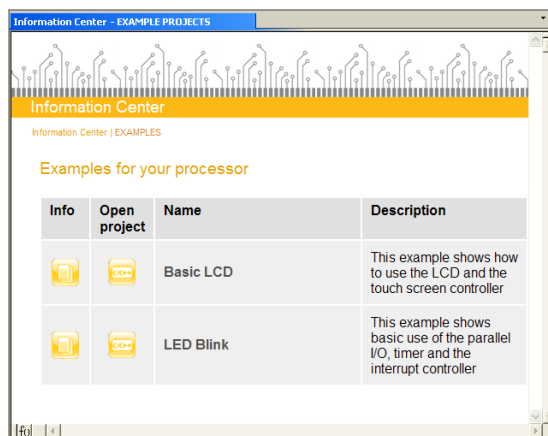
#### To run an example project:

- 1 Choose **Help>Information Center** and click **EXAMPLE PROJECTS**.
- 2 Click the download button for the chip manufacturer that matches your device.
- 3 In the dialog box that is displayed, choose where to get the examples from. Choose between:
  - Download from IAR Systems

- Copy from the installation DVD. In this case, use the browse button to locate the required self extracting example archive. You can find the archive in the `\examples-archive` directory on the DVD.

The examples for the selected device vendor will be extracted to your computer (in the `Program Data` directory or the corresponding directory depending on your Windows operating system).

- 4 In the list of downloaded examples, click the chip manufacturer and browse to the specific evaluation board or starter kit you are using.



- 5 Click the **Open Project** button.
- 6 In the dialog box that appears, choose a destination folder for your project.
- 7 The available example projects are displayed in the workspace window. Select one of the projects, and if it is not the active project (highlighted in bold), right-click it and choose **Set As Active** from the context menu.
- 8 To view the project settings, select the project and choose **Options** from the context menu. Verify the settings for **General Options>Target>Processor configuration** and **Debugger>Setup>Driver**. As for other settings, the project is set up to suit the target system you selected.

For more information about the C-SPY options and how to configure C-SPY to interact with the target board, see *Debugger options*, page 293.

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



- 9 To compile and link the application, choose **Project>Make** or click the **Make** button.

**I0** To start C-SPY, choose **Project>Debug** or click the **Download and Debug** button. If C-SPY fails to establish contact with the target system, see *Resolving problems*, page 342.



**I1** Choose **Debug>Go** or click the **Go** button to start the application.

Click the **Stop** button to stop execution.

---

## Reference information on starting C-SPY

Reference information about:

- *C-SPY Debugger main window*, page 59
- *Images window*, page 64
- *Get Alternative File dialog box*, page 65
- *Get Example Projects dialog box*, page 66

See also:

- Tools options for the debugger in the *IDE Project Management and Building Guide*.

## C-SPY Debugger main window

When you start a debug session, these debugger-specific items appear in the main IAR Embedded Workbench IDE window:

- A dedicated **Debug** menu with commands for executing and debugging your application
- Depending on the C-SPY driver you are using, a driver-specific menu, often referred to as the *Driver menu* in this documentation. Typically, this menu contains menu commands for opening driver-specific windows and dialog boxes.
- A special debug toolbar
- Several windows and dialog boxes specific to C-SPY.

The C-SPY main window might look different depending on which components of the product installation you are using.

### Menu bar

These menus are available during a debug session:

#### Debug

Provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

**C-SPY driver menu**

Provides commands specific to a C-SPY driver. The driver-specific menu is only available when the driver is used. For information about the driver-specific menu commands, see *Reference information on C-SPY driver menus*, page 329.

**Debug menu**

The **Debug** menu is available during a debug session. The **Debug** menu provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.



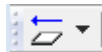
These commands are available:

**Go F5**

Executes from the current statement or instruction until a breakpoint or program exit is reached.

**Break**

Stops the application execution.

**Reset**

Resets the target processor. Click the drop-down button to access a menu with additional commands.

**Enable Run to 'label'**, where *label* typically is *main*. Enables and disables the project option **Run to** without exiting the debug session. This menu command is only available if you have selected **Run to** in the **Options** dialog box.

**Reset strategies**, which contains a list of reset strategies supported by the C-SPY driver you are using. This means that you can choose a different reset strategy than the one used initially without exiting the debug session. Reset strategies are only available if the C-SPY driver you are using supports alternate reset strategies.



### Stop Debugging (Ctrl+Shift+D)

Stops the debugging session and returns you to the project manager.



### Step Over (F10)

Executes the next statement, function call, or instruction, without entering C or C++ functions or assembler subroutines.



### Step Into (F11)

Executes the next statement or instruction, or function call, entering C or C++ functions or assembler subroutines.



### Step Out (Shift+F11)

Executes from the current statement up to the statement after the call to the current function.



### Next Statement

Executes directly to the next statement without stopping at individual function calls.



### Run to Cursor

Executes from the current statement or instruction up to a selected statement or instruction.

### Autostep

Displays a dialog box where you can customize and perform autostepping, see *Autostep settings dialog box*, page 86.

### Set Next Statement

Moves the program counter directly to where the cursor is, without executing any source code. Note, however, that this creates an anomaly in the program flow and might have unexpected effects.

### C++ Exceptions>

#### Break on Throw

This menu command is not supported by your product package.

### C++ Exceptions>

#### Break on Uncaught Exception

This menu command is not supported by your product package.

### **Memory>Save**

Displays a dialog box where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 150.

### **Memory>Restore**

Displays a dialog box where you can load the contents of a file in, for example Intel-extended or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 151.

### **Refresh**

Refreshes the contents of all debugger windows. Because window updates are automatic, this is needed only in unusual situations, such as when target memory is modified in ways C-SPY cannot detect. It is also useful if code that is displayed in the Disassembly window is changed.

### **Macros**

Displays a dialog box where you can list, register, and edit your macro files and functions, see *Using C-SPY macros*, page 209.

### **Logging>Set Log file**

Displays a dialog box where you can choose to log the contents of the **Debug Log** window to a file. You can select the type and the location of the log file. You can choose what you want to log: errors, warnings, system information, user messages, or all of these. See *Log File dialog box*, page 85.

### **Logging>**

#### **Set Terminal I/O Log file**

Displays a dialog box where you can choose to log simulated target access communication to a file. You can select the destination of the log file. See *Terminal I/O Log File dialog box*, page 83

## **C-SPY windows**

Depending on the C-SPY driver you are using, these windows specific to C-SPY are available during a debug session:

- C-SPY Debugger main window
- Disassembly window
- Memory window
- Symbolic Memory window
- Register window
- Watch window
- Locals window
- Auto window

- Quick Watch window
- Statics window
- Call Stack window
- Trace window
- Function Trace window
- Timeline window
- Terminal I/O window
- Code Coverage window
- Function Profiler window
- Images window
- Stack window
- Symbols window.

Additional windows are available depending on which C-SPY driver you are using.

### Editing in C-SPY windows

You can edit the contents of the **Memory**, **Symbolic Memory**, **Register**, **Auto**, **Watch**, **Locals**, **Statics**, and **Quick Watch** windows.

Use these keyboard keys to edit the contents of these windows:

<b>Enter</b>	Makes an item editable and saves the new value.
<b>Esc</b>	Cancels a new value.

In windows where you can edit the **Expression** field and in the **Quick Watch** window, you can specify the number of elements to be displayed in the field by adding a semicolon followed by an integer. For example, to display only the three first elements of an array named `myArray`, or three elements in sequence starting with the element pointed to by a pointer, write:

```
myArray;3
```

To display three elements pointed to by `myPtr`, `myPtr+1`, and `myPtr+2`, write:

```
myPtr;3
```

Optionally, add a comma and another integer that specifies which element to start with. For example, to display elements 10–14, write:

```
myArray;5,10
```

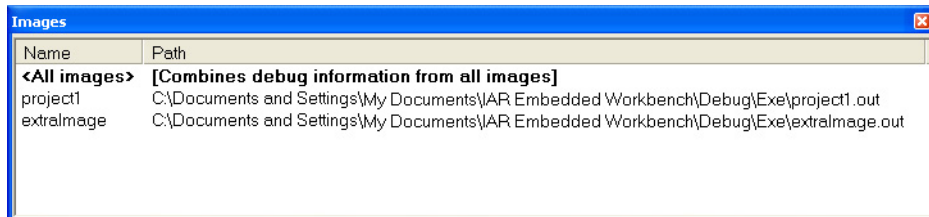
To display `myPtr+10`, `myPtr+11`, `myPtr+12`, `myPtr+13`, and `myPtr+14`, write:

```
myPtr;5,10
```

**Note:** For pointers, there are no built-in limits on displayed element count, and no validation of the pointer value.

## Images window

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



This window lists all currently loaded images (debug files).

Normally, a source application consists of exactly one image that you debug. However, you can also load additional images. This means that the complete debuggable unit consists of several images.

### Requirements

None; this window is always available.

### Display area

C-SPY can either use debug information from all of the loaded images simultaneously, or from one image at a time. Double-click on a row to show information only for that image. The current choice is highlighted.

This area lists the loaded images in these columns:

#### Name

The name of the loaded image.

#### Path

The path to the loaded image.



## Context menu

This context menu is available:



These commands are available:

### Show all images

Shows debug information for all loaded debug images.

### Show only *image*

Shows debug information for the selected debug image.

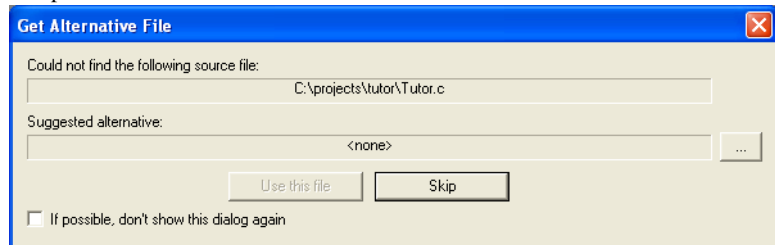
## Related information

For related information, see:

- *Loading multiple images*, page 55
- *Images*, page 296
- `__loadImage`, page 231.

## Get Alternative File dialog box

The **Get Alternative File** dialog box is displayed if C-SPY cannot automatically find the source files to be loaded, for example if the application was built on another computer.



### Could not find the following source file

The missing source file.

### Suggested alternative

Specify an alternative file.

### Use this file

After you have specified an alternative file, **Use this file** establishes that file as the alias for the requested file. Note that after you have chosen this action, C-SPY will automatically locate other source files if these files reside in a directory structure similar to the first selected alternative file.

The next time you start a debug session, the selected alternative file will be preloaded automatically.

### Skip

C-SPY will assume that the source file is not available for this debug session.

### If possible, don't show this dialog again

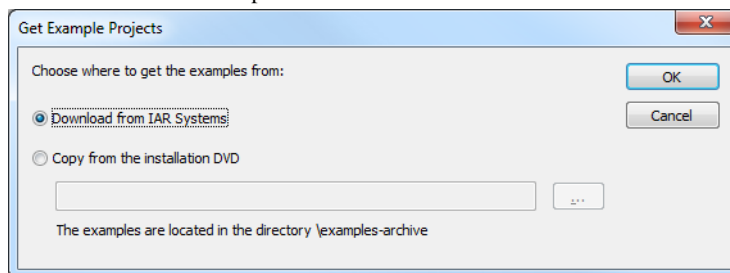
Instead of displaying the dialog box again for a missing source file, C-SPY will use the previously supplied response.

### Related information

For related information, see *Starting a debug session with source files missing*, page 54.

## Get Example Projects dialog box

The **Get Example Projects** dialog box is displayed when you have clicked the download button for a chip manufacturer in the IAR Information Center.



See also, *Running example projects*, page 57.

### Download from IAR Systems

Downloads the application from IAR Systems.

**Copy from the installation DVD**

Copies the application from the installation DVD. In this case, use the browse button to locate the required self extracting example archive. You can find the archive in the `\examples-archive` directory on the DVD.

The examples for the selected device vendor will be extracted to your computer (in the `Program Data` directory or the corresponding directory depending on your Windows operating system).



# Executing your application

- Introduction to application execution
- Reference information on application execution

---

## Introduction to application execution

These topics are covered:

- Briefly about application execution
- Source and disassembly mode debugging
- Single stepping
- Stepping speed
- Running the application
- Highlighting
- Call stack information
- Terminal input and output
- Debug logging

### **BRIEFLY ABOUT APPLICATION EXECUTION**

C-SPY allows you to monitor and control the execution of your application. By single-stepping through it, and setting breakpoints, you can examine details about the application execution, for example the values of variables and registers. You can also use the call stack to step back and forth in the function call chain.

The terminal I/O and debug log features let you interact with your application.

You can find commands for execution on the **Debug** menu and on the toolbar.

### **SOURCE AND DISASSEMBLY MODE DEBUGGING**

C-SPY allows you to switch between source mode and disassembly mode debugging as needed.

Source debugging provides the fastest and easiest way of developing your application, without having to worry about how the compiler or assembler has implemented the code. In the editor windows you can execute the application one statement at a time while monitoring the values of variables and data structures.

Disassembly mode debugging lets you focus on the critical sections of your application, and provides you with precise control of the application code. You can open a disassembly window which displays a mnemonic assembler listing of your application based on actual memory contents rather than source code, and lets you execute the application exactly one machine instruction at a time.

Regardless of which mode you are debugging in, you can display registers and memory, and change their contents.

## SINGLE STEPPING

C-SPY allows more stepping precision than most other debuggers because it is not line-oriented but statement-oriented. The compiler generates detailed stepping information in the form of *step points* at each statement, and at each function call. That is, source code locations where you might consider whether to execute a step into or a step over command. Because the step points are located not only at each statement but also at each function call, the step functionality allows a finer granularity than just stepping on statements.

There are several factors that can slow down the stepping speed. If you find it too slow, see *Slow stepping speed*, page 344 for some tips.

### The step commands

There are four step commands:

- **Step Into**
- **Step Over**
- **Next Statement**
- **Step Out.**

Using the **Autostep settings** dialog box, you can automate the single stepping. For more information, see *Autostep settings dialog box*, page 86.

Consider this example and assume that the previous step has taken you to the  $f(i)$  function call (highlighted):

```
extern int g(int);
int f(int n)
{
    value = g(n-1) + g(n-2) + g(n-3);
    return value;
}
int main()
{
    ...
    f(i);
    value ++;
}
```



### Step Into

While stepping, you typically consider whether to step into a function and continue stepping inside the function or subroutine. The **Step Into** command takes you to the first step point within the subroutine  $g(n-1)$ :

```
extern int g(int);
int f(int n)
{
    value = g(n-1) + g(n-2) + g(n-3);
    return value;
}
```

The **Step Into** command executes to the next step point in the normal flow of control, regardless of whether it is in the same or another function.



### Step Over

The **Step Over** command executes to the next step point in the same function, without stopping inside called functions. The command would take you to the  $g(n-2)$  function call, which is not a statement on its own but part of the same statement as  $g(n-1)$ . Thus, you can skip uninteresting calls which are parts of statements and instead focus on critical parts:

```
extern int g(int);
int f(int n)
{
    value = g(n-1) + g(n-2) + g(n-3);
    return value;
}
```



### Next Statement

The **Next Statement** command executes directly to the next statement, in this case `return value`, allowing faster stepping:

```
extern int g(int);
int f(int n)
{
    value = g(n-1) + g(n-2) + g(n-3);
    return value;
}
```



### Step Out

When inside the function, you can—if you wish—use the **Step Out** command to step out of it before it reaches the exit. This will take you directly to the statement immediately after the function call:

```
extern int g(int);
int f(int n)
{
    value = g(n-1) + g(n-2) g(n-3);
    return value;
}
int main()
{
    ...
    f(i);
    value ++;
}
```

The possibility of stepping into an individual function that is part of a more complex statement is particularly useful when you use C code containing many nested function calls. It is also very useful for C++, which tends to have many implicit function calls, such as constructors, destructors, assignment operators, and other user-defined operators.

This detailed stepping can in some circumstances be either invaluable or unnecessarily slow. For this reason, you can also step only on statements, which means faster stepping.

## STEPPING SPEED

Stepping in C-SPY is normally performed using breakpoints. When performing a step command, a breakpoint is set on the next statement and the program executes until reaching this breakpoint. If you are debugging using a hardware debugger system, the number of hardware breakpoints—typically used for setting a stepping breakpoint, at least in code that is located in flash/ROM memory—is limited. If you for example, step into a C `switch` statement, breakpoints are set on each branch, and hence, this might



consume several hardware breakpoints. If the number of available hardware breakpoints is exceeded, C-SPY switches into single stepping at assembly level, which can be very slow.

For this reason, it can be helpful to keep track of how many hardware breakpoints are used and make sure to some of them are left for stepping. For more information, see *Breakpoints in the C-SPY hardware debugger drivers*, page 116 and *Breakpoint consumers*, page 118.

In addition to limited hardware breakpoints, these issues might also affect stepping speed:

- If Trace or Function profiling is enabled. This might slow down stepping because collected Trace data is processed after each step. Note that it is not sufficient to close the corresponding windows to disable Trace data collection. Instead, you must disable the **Enable/Disable** button in both the Trace and the Function profiling windows.
- If the **Register** window is open and displays SFR registers. This might slow down stepping because all registers in the selected register group must be read from the hardware after each step. To solve this, you can choose to view only a limited selection of SFR register; you can choose between two alternatives. Either type `#SFR_name` (where `#SFR_name` reflects the name of the SFR you want to monitor) in the **Watch** window, or create your own filter for displaying a limited group of SFRs in the **Register** window. See *Defining application-specific register groups*, page 144.
- If any of the **Memory** or **Symbolic** memory windows is open. This might slow down stepping because the visible memory must be read after each step.
- If any of the expression related windows such as **Watch**, **Locals**, **Statics** is open. This might slow down stepping speed because all these windows reads memory after each step.
- If the **Stack** window is open and especially if the option **Enable graphical stack display and stack usage tracking** option is enabled. To disable this option, choose **Tools>Options>Stack** and disable it.
- If a too slow communication speed has been set up between C-SPY and the target board/emulator you should consider to increase the speed, if possible.

## RUNNING THE APPLICATION



### Go

The **Go** command continues execution from the current position until a breakpoint or program exit is reached.



## Run to Cursor

The **Run to Cursor** command executes to the position in the source code where you have placed the cursor. The **Run to Cursor** command also works in the **Disassembly** window and in the **Call Stack** window.

## HIGHLIGHTING

At each stop, C-SPY highlights the corresponding C or C++ source or instruction with a green color, in the editor and the **Disassembly** window respectively. In addition, a green arrow appears in the editor window when you step on C or C++ source level, and in the **Disassembly** window when you step on disassembly level. This is determined by which of the windows is the active window. If none of the windows are active, it is determined by which of the windows was last active.

```
Tutor.c Utilities.c
void init_fib( void )
{
  int i = 45;
  root[0] = root[1] = 1;
  for ( i=2 ; i<MAX_FIB ; i++)
  {
```

For simple statements without function calls, the whole statement is typically highlighted. When stopping at a statement with function calls, C-SPY highlights the first call because this illustrates more clearly what **Step Into** and **Step Over** would mean at that time.

Occasionally, you will notice that a statement in the source window is highlighted using a pale variant of the normal highlight color. This happens when the program counter is at an assembler instruction which is part of a source statement but not exactly at a step point. This is often the case when stepping in the **Disassembly** window. Only when the program counter is at the first instruction of the source statement, the ordinary highlight color is used.

## CALL STACK INFORMATION

The compiler generates extensive backtrace information. This allows C-SPY to show, without any runtime penalty, the complete function call chain at any time.



Typically, this is useful for two purposes:

- Determining in what context the current function has been called
- Tracing the origin of incorrect values in variables and in parameters, thus locating the function in the call chain where the problem occurred.

The **Call Stack** window shows a list of function calls, with the current function at the top. When you inspect a function in the call chain, the contents of all affected windows

are updated to display the state of that particular call frame. This includes the editor, **Locals**, **Register**, **Watch** and **Disassembly** windows. A function would normally not make use of all registers, so these registers might have undefined states and be displayed as dashes (---).

In the editor and **Disassembly** windows, a green highlight indicates the topmost, or current, call frame; a yellow highlight is used when inspecting other frames.

For your convenience, it is possible to select a function in the call stack and click the **Run to Cursor** command to execute to that function.

Assembler source code does not automatically contain any backtrace information. To see the call chain also for your assembler modules, you can add the appropriate `CFI` assembler directives to the assembler source code. For further information, see the *AVR® IAR Assembler Reference Guide*.

## TERMINAL INPUT AND OUTPUT

Sometimes you might have to debug constructions in your application that use `stdin` and `stdout` without an actual hardware device for input and output. The **Terminal I/O** window lets you enter input to your application, and display output from it. You can also direct terminal I/O to a file, using the **Terminal I/O Log Files** dialog box.



This facility is useful in two different contexts:

- If your application uses `stdin` and `stdout`
- For producing debug trace printouts.

For more information, see *Terminal I/O window*, page 82 and *Terminal I/O Log File dialog box*, page 83.

## DEBUG LOGGING

The **Debug Log** window displays debugger output, such as diagnostic messages, macro-generated output, event log messages, and information about trace.



It can sometimes be convenient to log the information to a file where you can easily inspect it. The two main advantages are:

- The file can be opened in another tool, for instance an editor, so you can navigate and search within the file for particularly interesting parts
- The file provides history about how you have controlled the execution, for instance, which breakpoints that have been triggered etc.

## Reference information on application execution

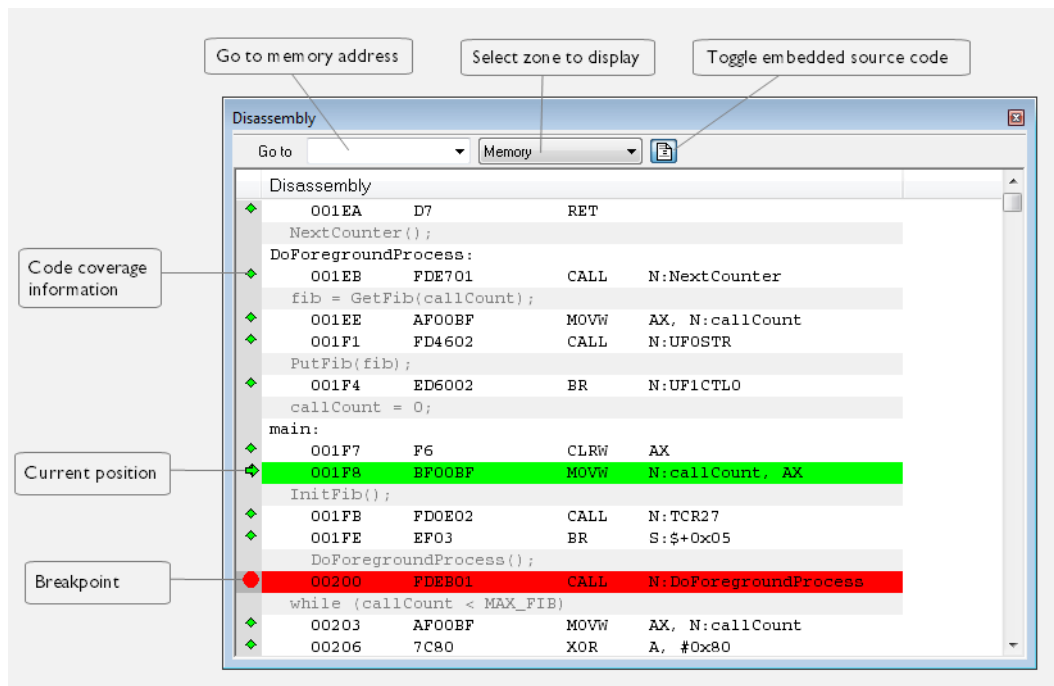
Reference information about:

- *Disassembly window*, page 76
- *Call Stack window*, page 80
- *Terminal I/O window*, page 82
- *Terminal I/O Log File dialog box*, page 83
- *Debug Log window*, page 84
- *Log File dialog box*, page 85
- *Report Assert dialog box*, page 86
- *Autostep settings dialog box*, page 86

See also Terminal I/O options in the *IDE Project Management and Building Guide*.

## Disassembly window

The C-SPY **Disassembly** window is available from the **View** menu.



This window shows the application being debugged as disassembled application code.

**To change the default color of the source code in the Disassembly window:**

- 1 Choose **Tools>Options>Debugger**.
- 2 Set the default color using the **Source code coloring in disassembly window** option.



To view the corresponding assembler code for a function, you can select it in the editor window and drag it to the **Disassembly** window.

**Requirements**

None; this window is always available.

**Toolbar**

The toolbar contains:

**Go to**

The memory location or symbol you want to view.

**Zone**

Selects a memory zone, see *C-SPY memory zones*, page 142.

**Toggle Mixed-Mode**

Toggles between displaying only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information

**Display area**

The display area shows the disassembled application code.

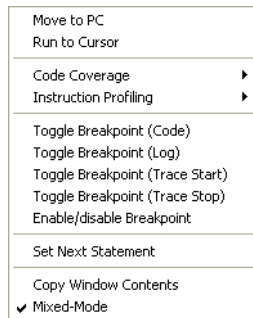
This area contains these graphic elements:

Green highlight	Indicates the current position, that is the next assembler instruction to be executed. To move the cursor to any line in the <b>Disassembly</b> window, click the line. Alternatively, move the cursor using the navigation keys.
Yellow highlight	Indicates a position other than the current position, such as when navigating between frames in the <b>Call Stack</b> window or between items in the <b>Trace</b> window.
Red dot	Indicates a breakpoint. Double-click in the gray left-side margin of the window to set a breakpoint. For more information, see <i>Breakpoints</i> , page 113.
Green diamond	Indicates code that has been executed—that is, code coverage.

If instruction profiling has been enabled from the context menu, an extra column in the left-side margin appears with information about how many times each instruction has been executed.

### Context menu

This context menu is available:



**Note:** The contents of this menu are dynamic, which means that the commands on the menu might depend on your product package.

These commands are available:

#### Move to PC

Displays code at the current program counter location.

**Run to Cursor**

Executes the application from the current position up to the line containing the cursor.

**Code Coverage**

Displays a submenu that provides commands for controlling code coverage. This command is only enabled if the driver you are using supports it.

<b>Enable</b>	Toggles code coverage on or off.
<b>Show</b>	Toggles the display of code coverage on or off. Executed code is indicated by a green diamond.
<b>Clear</b>	Clears all code coverage information.

**Instruction Profiling**

Displays a submenu that provides commands for controlling instruction profiling. This command is only enabled if the driver you are using supports it.

<b>Enable</b>	Toggles instruction profiling on or off.
<b>Show</b>	Toggles the display of instruction profiling on or off. For each instruction, the left-side margin displays how many times the instruction has been executed.
<b>Clear</b>	Clears all instruction profiling information.

**Toggle Breakpoint (Code)**

Toggles a code breakpoint. Assembler instructions and any corresponding label at which code breakpoints have been set are highlighted in red. For more information, see *Code breakpoints dialog box*, page 127.

**Toggle Breakpoint (Log)**

Toggles a log breakpoint for trace printouts. Assembler instructions at which log breakpoints have been set are highlighted in red. For more information, see *Log breakpoints dialog box*, page 129.

**Toggle Breakpoint (Trace Start)**

Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Start breakpoints dialog box*, page 178.

**Toggle Breakpoint (Trace Stop)**

Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Stop breakpoints dialog box*, page 179.

**Enable/Disable Breakpoint**

Enables and Disables a breakpoint. If there is more than one breakpoint at a specific line, all those breakpoints are affected by the **Enable/Disable** command.

**Edit Breakpoint**

Displays the breakpoint dialog box to let you edit the currently selected breakpoint. If there is more than one breakpoint on the selected line, a submenu is displayed that lists all available breakpoints on that line.

**Set Next Statement**

Sets the program counter to the address of the instruction at the insertion point.

**Copy Window Contents**

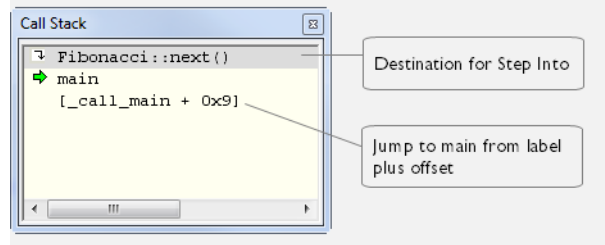
Copies the selected contents of the **Disassembly** window to the clipboard.

**Mixed-Mode**

Toggles between showing only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

## Call Stack window

The **Call Stack** window is available from the **View** menu.



This window displays the C function call stack with the current function at the top. To inspect a function call, double-click it. C-SPY now focuses on that call frame instead.



If the next **Step Into** command would step to a function call, the name of the function is displayed in the grey bar at the top of the window. This is especially useful for implicit function calls, such as C++ constructors, destructors, and operators.

### Requirements

None; this window is always available.

### Display area

Provided that the command **Show Arguments** is enabled, each entry in the display area has the format:

```
function(values)***
```

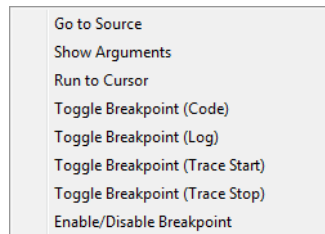
where

*(values)* is a list of the current values of the parameters, or empty if the function does not take any parameters.

\*\*\*, if visible, indicates that the function has been inlined by the compiler. For information about function inlining, see the *IAR C/C++ Compiler Reference Guide for AVR*.

### Context menu

This context menu is available:



These commands are available:

#### Go to Source

Displays the selected function in the **Disassembly** or editor windows.

#### Show Arguments

Shows function arguments.

#### Run to Cursor

Executes until return to the function selected in the call stack.

#### Toggle Breakpoint (Code)

Toggles a code breakpoint.

**Toggle Breakpoint (Log)**

Toggles a log breakpoint.

**Toggle Breakpoint (Trace Start)**

Toggles a Trace Start breakpoint. When the breakpoint is triggered, trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports it.

**Toggle Breakpoint (Trace Stop)**

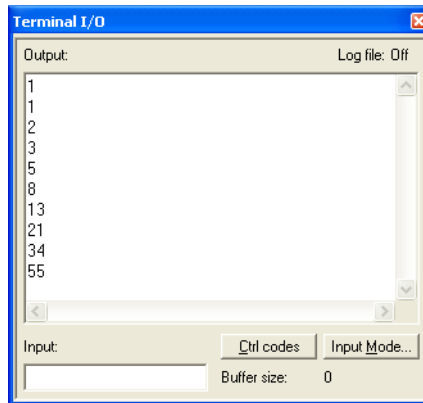
Toggles a Trace Stop breakpoint. When the breakpoint is triggered, trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports it.

**Enable/Disable Breakpoint**

Enables or disables the selected breakpoint

## Terminal I/O window

The **Terminal I/O** window is available from the **View** menu.



Use this window to enter input to your application, and display output from it.

**To use this window, you must:**

- I Link your application with the option **With I/O emulation modules**.

C-SPY will then direct `stdin`, `stdout` and `stderr` to this window. If the **Terminal I/O** window is closed, C-SPY will open it automatically when input is required, but not for output.

**Requirements**

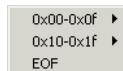
None; this window is always available.

**Input**

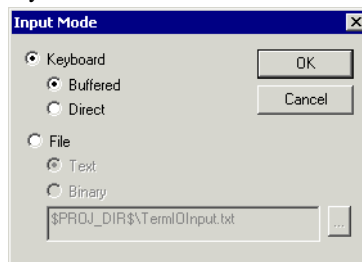
Type the text that you want to input to your application.

**Ctrl codes**

Opens a menu for input of special characters, such as EOF (end of file) and NUL.

**Input Mode**

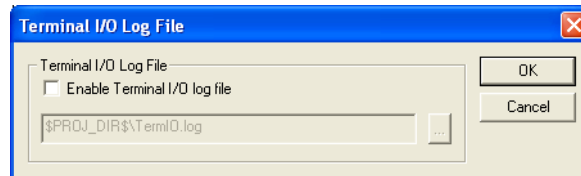
Opens the **Input Mode** dialog box where you choose whether to input data from the keyboard or from a file.



For reference information about the options available in this dialog box, see Terminal I/O options in *IDE Project Management and Building Guide*.

**Terminal I/O Log File dialog box**

The **Terminal I/O Log File** dialog box is available by choosing **Debug>Logging>Set Terminal I/O Log File**.



Use this dialog box to select a destination log file for terminal I/O from C-SPY.

**Requirements**

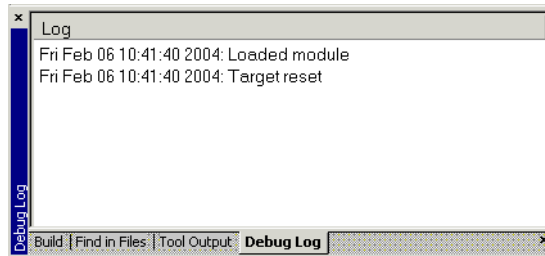
None; this dialog box is always available.

## Terminal IO Log Files

Controls the logging of terminal I/O. To enable logging of terminal I/O to a file, select **Enable Terminal IO log file** and specify a filename. The default filename extension is `.log`. A browse button is available for your convenience.

## Debug Log window

The **Debug Log** window is available by choosing **View>Messages**.



This window displays debugger output, such as diagnostic messages, macro-generated output, event log messages, and information about trace. This output is only available during a debug session. When opened, this window is, by default, grouped together with the other message windows, see *IDE Project Management and Building Guide*.

Double-click any rows in one of the following formats to display the corresponding source code in the editor window:

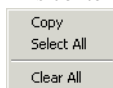
```
<path> (<row>) :<message>
<path> (<row>, <column>) :<message>
```

## Requirements

None; this window is always available.

## Context menu

This context menu is available:



These commands are available:

### Copy

Copies the contents of the window.

### Select All

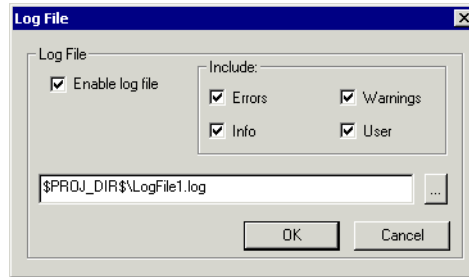
Selects the contents of the window.

**Clear All**

Clears the contents of the window.

**Log File dialog box**

The **Log File** dialog box is available by choosing **Debug>Logging>Set Log File**.



Use this dialog box to log output from C-SPY to a file.

**Requirements**

None; this dialog box is always available.

**Enable Log file**

Enables or disables logging to the file.

**Include**

The information printed in the file is, by default, the same as the information listed in the Log window. Use the browse button, to override the default file and location of the log file (the default filename extension is `log`). To change the information logged, choose between:

**Errors**

C-SPY has failed to perform an operation.

**Warnings**

An error or omission of concern.

**Info**

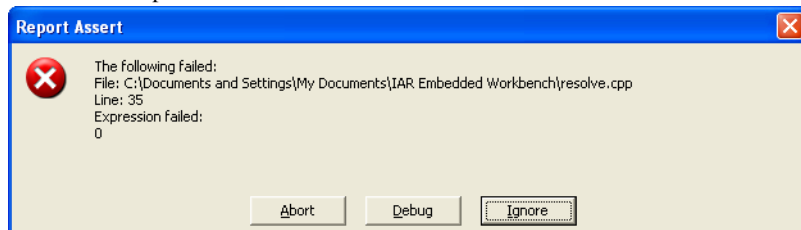
Progress information about actions C-SPY has performed.

**User**

Messages from C-SPY macros, that is, your messages using the `__message` statement.

## Report Assert dialog box

The **Report Assert dialog box** appears if you have a call to the `assert` function in your application source code, and the assert condition is false. In this dialog box you can choose how to proceed.



### Abort

The application stops executing and the runtime library function `abort`, which is part of your application on the target system, will be called. This means that the application itself terminates its execution.

### Debug

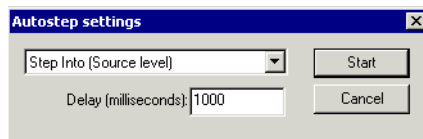
C-SPY stops the execution of the application and returns control to you.

### Ignore

The assertion is ignored and the application continues to execute.

## Autostep settings dialog box

The **Autostep settings** dialog box is available from the **Debug** menu.



Use this dialog box to customize autostepping.

The drop-down menu lists the available step commands.

### Requirements

None; this dialog box is always available.

### Delay

Specify the delay between each step in milliseconds.

# Variables and expressions

- Introduction to working with variables and expressions
- Working with variables and expressions
- Reference information on working with variables and expressions

---

## Introduction to working with variables and expressions

This section introduces different methods for looking at variables and introduces some related concepts.

These topics are covered:

- Briefly about working with variables and expressions
- C-SPY expressions
- Limitations on variable information.

### **BRIEFLY ABOUT WORKING WITH VARIABLES AND EXPRESSIONS**

There are several methods for looking at variables and calculating their values:

- **Tooltip watch**—in the editor window—provides the simplest way of viewing the value of a variable or more complex expressions. Just point at the variable with the mouse pointer. The value is displayed next to the variable.
- The **Auto** window displays a useful selection of variables and expressions in, or near, the current statement. The window is automatically updated when execution stops.
- The **Locals** window displays the local variables, that is, auto variables and function parameters for the active function. The window is automatically updated when execution stops.
- The **Watch** window allows you to monitor the values of C-SPY expressions and variables. The window is automatically updated when execution stops.
- The **Statics** window displays the values of variables with static storage duration. The window is automatically updated when execution stops.
- The **Macro Quicklaunch** window and the **Quick Watch** window give you precise control over when to evaluate an expression.

- The **Symbols** window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.
- The **Data Log** window and the **Data Log Summary** window display logs of accesses up to four different memory locations or areas you choose by setting Data Log breakpoints. Data logging can help you locate frequently accessed data. You can then consider whether you should place that data in more efficient memory.
- The Trace-related windows let you inspect the program flow up to a specific state. For more information, see *Trace*, page 165.

## C-SPY EXPRESSIONS

C-SPY expressions can include any type of C expression, except for calls to functions. The following types of symbols can be used in expressions:

- C/C++ symbols
- Assembler symbols (register names and assembler labels)
- C-SPY macro functions
- C-SPY macro variables.

Expressions that are built with these types of symbols are called C-SPY expressions and there are several methods for monitoring these in C-SPY. Examples of valid C-SPY expressions are:

```
i + j
i = 42
myVar = cVar
cVar = myVar + 2
#asm_label
#R2
#PC
my_macro_func(19)
```

If you have a static variable with the same name declared in several different functions, use the notation *function::variable* to specify which variable to monitor.

## C/C++ symbols

C symbols are symbols that you have defined in the C source code of your application, for instance variables, constants, and functions (functions can be used as symbols but cannot be executed). C symbols can be referenced by their names. Note that C++ symbols might implicitly contain function calls which are not allowed in C-SPY symbols and expressions.



**Note:** Some attributes available in C/C++, like `volatile`, are not fully supported by C-SPY. For example, this line will not be accepted by C-SPY:

```
sizeof(unsigned char volatile __memattr *)
```

However, this line will be accepted:

```
sizeof(unsigned char __memattr *)
```

## Assembler symbols

Assembler symbols can be assembler labels or registers, for example the program counter, the stack pointer, or other CPU registers. If a device description file is used, all memory-mapped peripheral units, such as I/O ports, can also be used as assembler symbols in the same way as the CPU registers. See *Modifying a device description file*, page 56.

Assembler symbols can be used in C-SPY expressions if they are prefixed by #.

Example	What it does
<code>#PC++</code>	Increments the value of the program counter.
<code>myVar = #SP</code>	Assigns the current value of the stack pointer register to your C-SPY variable.
<code>myVar = #label</code>	Sets <code>myVar</code> to the value of an integer at the address of <code>label</code> .
<code>myptr = &amp;#label7</code>	Sets <code>myptr</code> to an <code>int *</code> pointer pointing at <code>label7</code> .

Table 4: C-SPY assembler symbols expressions

In case of a name conflict between a hardware register and an assembler label, hardware registers have a higher precedence. To refer to an assembler label in such a case, you must enclose the label in back quotes ` (ASCII character 0x60). For example:

Example	What it does
<code>#PC</code>	Refers to the program counter.
<code>#`PC`</code>	Refers to the assembler label <code>PC</code> .

Table 5: Handling name conflicts between hardware registers and assembler labels

Which processor-specific symbols are available by default can be seen in the **Register** window, using the `CPU Registers` register group. See *Register window*, page 159.

## C-SPY macro functions

Macro functions consist of C-SPY macro variable definitions and macro statements which are executed when the macro is called.

For information about C-SPY macro functions and how to use them, see *Briefly about the macro language*, page 208.

## C-SPY macro variables

Macro variables are defined and allocated outside your application, and can be used in a C-SPY expression. In case of a name conflict between a C symbol and a C-SPY macro variable, the C-SPY macro variable will have a higher precedence than the C variable. Assignments to a macro variable assign both its value and type.

For information about C-SPY macro variables and how to use them, see *Reference information on the macro language*, page 214.

## Using sizeof

According to standard C, there are two syntactical forms of `sizeof`:

```
sizeof(type)
sizeof expr
```

The former is for types and the latter for expressions.

**Note:** In C-SPY, do not use parentheses around an expression when you use the `sizeof` operator. For example, use `sizeof x+2` instead of `sizeof (x+2)`.

## LIMITATIONS ON VARIABLE INFORMATION

The value of a C variable is valid only on step points, that is, the first instruction of a statement and on function calls. This is indicated in the editor window with a bright green highlight color. In practice, the value of the variable is accessible and correct more often than that.

When the program counter is inside a statement, but not at a step point, the statement or part of the statement is highlighted with a pale variant of the ordinary highlight color.

## Effects of optimizations

The compiler is free to optimize the application software as much as possible, as long as the expected behavior remains. The optimization can affect the code so that debugging might be more difficult because it will be less clear how the generated code relates to the source code. Typically, using a high optimization level can affect the code in a way that will not allow you to view a value of a variable as expected.

Consider this example:

```
myFunction()
{
  int i = 42;
  ...
  x = computer(i); /* Here, the value of i is known to C-SPY */
  ...
}
```

From the point where the variable `i` is declared until it is actually used, the compiler does not need to waste stack or register space on it. The compiler can optimize the code, which means that C-SPY will not be able to display the value until it is actually used. If you try to view the value of a variable that is temporarily unavailable, C-SPY will display the text:

```
Unavailable
```

If you need full information about values of variables during your debugging session, you should make sure to use the lowest optimization level during compilation, that is, **None**.

---

## Working with variables and expressions

These tasks are covered:

- Using the windows related to variables and expressions
- Viewing assembler variables
- Getting started using data logging

### USING THE WINDOWS RELATED TO VARIABLES AND EXPRESSIONS

Where applicable, you can add, modify, and remove expressions, and change the display format in the windows related to variables and expressions.

To add a value you can also click in the dotted rectangle and type the expression you want to examine. To modify the value of an expression, click the **Value** field and modify its content. To remove an expression, select it and press the Delete key.



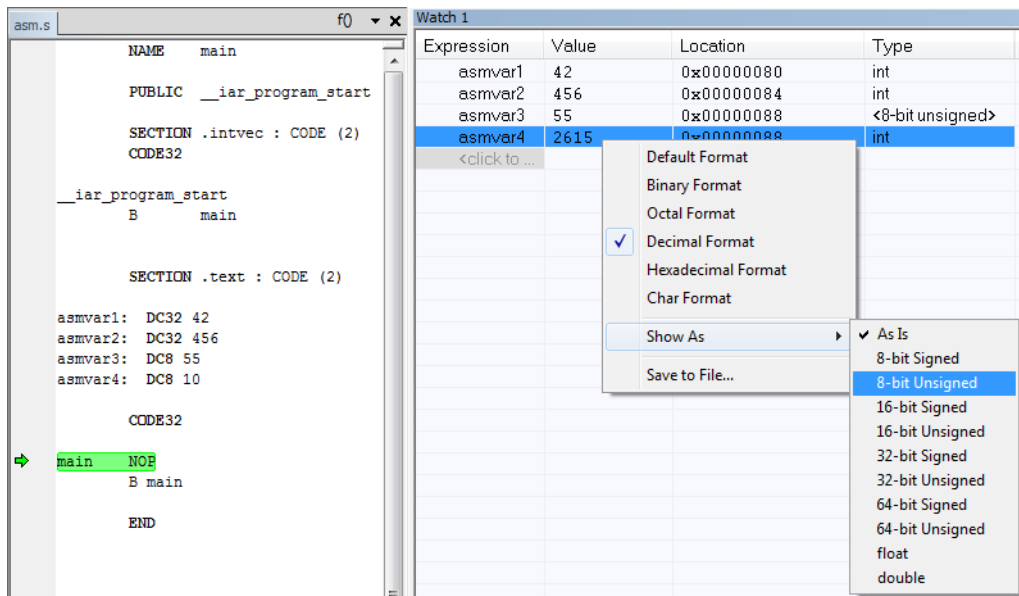
For text that is too wide to fit in a column—in any of the these windows, except the **Trace** window—and thus is truncated, just point at the text with the mouse pointer and tooltip information is displayed.

Right-click in any of the windows to access the context menu which contains additional commands. Convenient drag-and-drop between windows is supported, except for in the **Locals** window, Data logging windows, and the **Quick Watch** window where it is not relevant.

### VIEWING ASSEMBLER VARIABLES

An assembler label does not convey any type information at all, which means C-SPY cannot easily display data located at that label without getting extra information. To view data conveniently, C-SPY by default treats all data located at assembler labels as variables of type `int`. However, in the **Watch**, and **Quick Watch** windows, you can select a different interpretation to better suit the declaration of the variables.

In this figure, you can see four variables in the **Watch** window and their corresponding declarations in the assembler source file to the left:



Note that `asmvar4` is displayed as an `int`, although the original assembler declaration probably intended for it to be a single byte quantity. From the context menu you can make C-SPY display the variable as, for example, an 8-bit unsigned variable. This has already been specified for the `asmvar3` variable.

## GETTING STARTED USING DATA LOGGING

- 1 In the **Breakpoints** or **Memory** window, right-click and choose **New Breakpoints>Data Log** to open the breakpoints dialog box. Set a Data Log breakpoint on the data that you want to collect log information for. You can set up to four data log breakpoints.
- 2 Choose **C-SPY driver>Data Log** to open the **Data Log** window. Optionally, you can also choose:
  - **C-SPY driver>Data Log Summary** to open the **Data Log Summary** window
  - **C-SPY driver>Timeline** to open the **Timeline** window to view the Data Log graph.
- 3 From the context menu, available in the **Data Log** window, choose **Enable** to enable the logging.

- 4 Start executing your application program to collect the log information.
- 5 To view the data log information, look in any of the **Data Log** window, **Data Log Summary** window, or the Data graph in the **Timeline** window.
- 6 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
- 7 To disable data logging, choose **Disable** from the context menu in each window where you have enabled it.

---

## Reference information on working with variables and expressions

Reference information about:

- *Auto window*, page 94
- *Locals window*, page 95
- *Watch window*, page 97
- *Statics window*, page 99
- *Quick Watch window*, page 102
- *Symbols window*, page 104
- *Resolve Symbol Ambiguity dialog box*, page 106
- *Data Log window*, page 107
- *Data Log Summary window*, page 109

See also:

- *Reference information on trace*, page 168 for trace-related reference information
- *Macro Quicklaunch window*, page 262

## Auto window

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

Expression	Value	Location	Type
i	5	0x7	short
Fib[i]	0	Memory:0xC00C	unsigned int
Fib	<array>	Memory:0xC002	unsigned int[10]
GetFib	GetFib (0xBC)		unsigned int(*)...

This window displays a useful selection of variables and expressions in, or near, the current statement. Every time execution in C-SPY stops, the values in the **Auto** window are recalculated. Values that have changed since the last stop are highlighted in red.

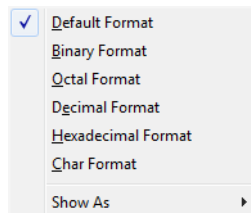
For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 59.

### Requirements

None; this window is always available.

### Context menu

This context menu is available:



These commands are available:

**Default Format,**  
**Binary Format,**  
**Octal Format,**  
**Decimal Format,**  
**Hexadecimal Format,**  
**Char Format**

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

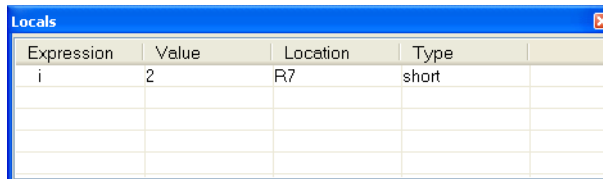
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

### Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 91.

## Locals window

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



Expression	Value	Location	Type
i	2	R7	short

This window displays the local variables and parameters for the current function. Every time execution in C-SPY stops, the values in the window are recalculated. Values that have changed since the last stop are highlighted in red.

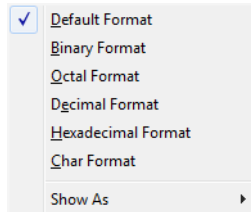
For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 59.

### Requirements

None; this window is always available.

## Context menu

This context menu is available:



These commands are available:

**Default Format,  
Binary Format,  
Octal Format,  
Decimal Format,  
Hexadecimal Format,  
Char Format**

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

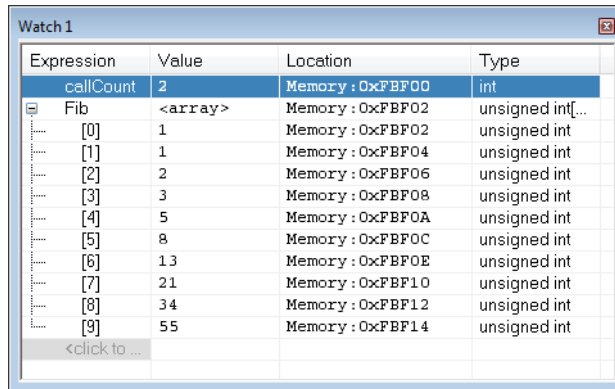
### Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 91.



## Watch window

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



Expression	Value	Location	Type
callCount	2	Memory : 0xFBFF00	int
Fib	<array>	Memory : 0xFBFF02	unsigned int[...]
[0]	1	Memory : 0xFBFF02	unsigned int
[1]	1	Memory : 0xFBFF04	unsigned int
[2]	2	Memory : 0xFBFF06	unsigned int
[3]	3	Memory : 0xFBFF08	unsigned int
[4]	5	Memory : 0xFBFF0A	unsigned int
[5]	8	Memory : 0xFBFF0C	unsigned int
[6]	13	Memory : 0xFBFF0E	unsigned int
[7]	21	Memory : 0xFBFF10	unsigned int
[8]	34	Memory : 0xFBFF12	unsigned int
[9]	55	Memory : 0xFBFF14	unsigned int
<click to ...>			

Use this window to monitor the values of C-SPY expressions or variables. You can open up to four instances of this window, where you can view, add, modify, and remove expressions. Tree structures of arrays, structs, and unions are expandable, which means that you can study each item of these.

Every time execution in C-SPY stops, the values in the **Watch** window are recalculated. Values that have changed since the last stop are highlighted in red.



Be aware that expanding very huge arrays can cause an out-of-memory crash. To avoid this, expansion is automatically performed in steps of 5000 elements.

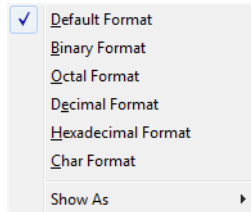
For more information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 59.

### Requirements

None; this window is always available.

## Context menu

This context menu is available:



These commands are available:

**Default Format,  
Binary Format,  
Octal Format,  
Decimal Format,  
Hexadecimal Format,  
Char Format**

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

### Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 91.

## Statics window

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

Variable	Value	Location	Type	Module
f <CppTutor\>	<class>	0x00000000	class std::ctype<char>	CppTutor
...	...	...	...	...
__vptr	0x20000A90	0x00000000	void (* const*)()	
f <CppTutor\>	<class>	0x200002F4	class std::num_punct<char>	CppTutor
f <CppTutor\>	<class>	0x20000308	class std::num_put<char, std::o...	CppTutor
msFib <Fibonacci\Fibonacci::msFib>	<array>	0x2000032C	unsigned long[100]	Fibonacci
[0]	1	0x2000032C	unsigned long	
[1]	1	0x20000330	unsigned long	
[2]	2	0x20000334	unsigned long	

This window displays the values of variables with static storage duration that you have selected. Typically, that is variables with file scope but it can also be static variables in functions and classes. Note that `volatile` declared variables with static storage duration will not be displayed.

Every time execution in C-SPY stops, the values in the **Statics** window are recalculated. Values that have changed since the last stop are highlighted in red.

Click any column header (except for **Value**) to sort on that column.

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 59.

### To select variables to monitor:

- 1 In the window, right-click and choose **Select statics** from the context menu. The window now lists all variables with static storage duration.
- 2 Either individually select the variables you want to display, or choose one of the **Select** commands from the context menu.
- 3 When you have made your selections, choose **Select statics** from the context menu to toggle back to normal display mode.

### Requirements

None; this window is always available.

## Display area

This area contains these columns:

### Expression

The name of the variable. The base name of the variable is followed by the full name, which includes module, class, or function scope. This column is not editable.

### Value

The value of the variable. Values that have changed are highlighted in red.

Dragging text or a variable from another window and dropping it on the **Value** column will assign a new value to the variable in that row.

This column is editable.

### Location

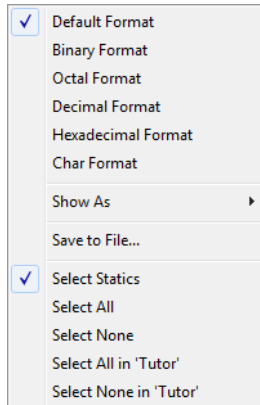
The location in memory where this variable is stored.

### Type

The data type of the variable.

## Context menu

This context menu is available:



These commands are available:

### **Default Format, Binary Format, Octal Format, Decimal Format, Hexadecimal Format, Char Format**

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

### **Save to File**

Saves the content of the **Statics** window to a log file.

**Select Statics**

Selects all variables with static storage duration; this command also enables all **Select** commands below. Select the variables you want to monitor. When you have made your selections, select this menu command again to toggle back to normal display mode.

**Select All**

Selects all variables.

**Select None**

Deselects all variables.

**Select All in *module***

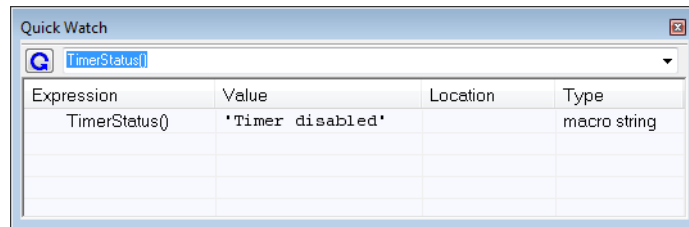
Selects all variables in the selected module.

**Select None in *module***

Deselects all variables in the selected module.

## Quick Watch window

The **Quick Watch** window is available from the **View** menu and from the context menu in the editor window.



Use this window to watch the value of a variable or expression and evaluate expressions at a specific point in time.

In contrast to the **Watch** window, the **Quick Watch** window gives you precise control over when to evaluate the expression. For single variables this might not be necessary, but for expressions with possible side effects, such as assignments and C-SPY macro functions, it allows you to perform evaluations under controlled conditions.

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 59.

**To evaluate an expression:**

- I In the editor window, right-click on the expression you want to examine and choose **Quick Watch** from the context menu that appears.

- 2 The expression will automatically appear in the **Quick Watch** window.

Alternatively:

- 3 In the **Quick Watch** window, type the expression you want to examine in the **Expressions** text box.



- 4 Click the **Recalculate** button to calculate the value of the expression.

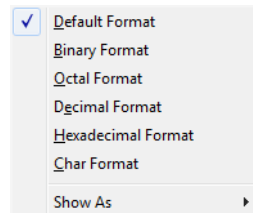
For an example, see *Using C-SPY macros*, page 209.

## Requirements

None; this window is always available.

## Context menu

This context menu is available:



These commands are available:

**Default Format,**  
**Binary Format,**  
**Octal Format,**  
**Decimal Format,**  
**Hexadecimal Format,**  
**Char Format**

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.

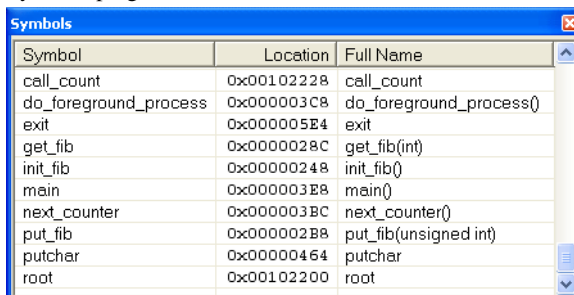
Structure fields      All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

### Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 91.

## Symbols window

The **Symbols** window is available from the **View** menu after you have enabled the Symbols plugin module.



Symbol	Location	Full Name
call_count	0x00102228	call_count
do_foreground_process	0x000003C8	do_foreground_process()
exit	0x000005E4	exit
get_fib	0x0000028C	get_fib(int)
init_fib	0x00000248	init_fib()
main	0x000003E8	main()
next_counter	0x000003BC	next_counter()
put_fib	0x000002B8	put_fib(unsigned int)
putchar	0x00000464	putchar
root	0x00102200	root

This window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.

To enable the Symbols plugin module, choose **Project>Options>Debugger>Select plugins to load>Symbols**.

### Requirements

None; this window is always available.

### Display area

This area contains these columns:

#### Symbol

The symbol name.



**Location**

The memory address.

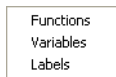
**Full name**

The symbol name; often the same as the contents of the Symbol column but differs for example for C++ member functions.

Click the column headers to sort the list by symbol name, location, or full name.

**Context menu**

This context menu is available:



These commands are available:

**Functions**

Toggles the display of function symbols on or off in the list.

**Variables**

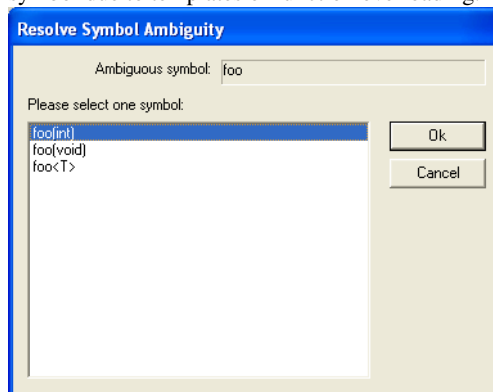
Toggles the display of variables on or off in the list.

**Labels**

Toggles the display of labels on or off in the list.

## Resolve Symbol Ambiguity dialog box

The **Resolve Symbol Ambiguity** dialog box appears, for example, when you specify a symbol in the Disassembly window to go to, and there are several instances of the same symbol due to templates or function overloading.



### Requirements

None; this window is always available.

### Ambiguous symbol

Indicates which symbol that is ambiguous.

### Please select one symbol

A list of possible matches for the ambiguous symbol. Select the one you want to use.

## Data Log window

The **Data Log** window is available from the C-SPY driver menu.

Time	Program Counter	I1	Address	s2	Address
0.160us	---			W 0x0000	@ 0x2004
0.160us	0xFFE00049	-	@ 0x2000		
24.480us	0xFFE000B5			R 0x0000	@ 0x2006
24.720us	0xFFE000BF			W 0x0042	@ 0x2004
24.760us	0xFFE000C6			R 0x0042	@ 0x2006
24.960us	0xFFE000E4	W 0x00004444	@ 0x2000		
<i>78.760us</i>	0xFFE00104			R 0x0042	@ 0x2004+?
79.000us	---			W 0x0084	@ 0x2004
100.800us	0xFFE00104			R 0x0084	@ 0x2006
101.040us	0xFFE0010E			W 0x00C6	@ 0x2004
<i>136.640us</i>	<b>Overflow</b>				
136.880us	0xFFE0010E			-	@ 0x2004

White rows indicate read accesses

Grey rows indicate write accesses

Use this window to log accesses to up to four different memory locations or areas.

See also *Getting started using data logging*, page 92.

### Requirements

The C-SPY simulator.

### Display area

Each row in the display area shows the time, the program counter, and, for every tracked data object, its value and address in these columns:

#### Time

If the time is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it.

This column is available when you have selected **Show time** from the context menu.

#### Cycles

The number of cycles from the start of the execution until the event. This information is cleared at reset.

If a cycle is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it.

This column is available when you have selected **Show cycles** from the context menu.

### Program Counter\*

Displays one of these:

An address, which is the content of the PC, that is, the address of the instruction that performed the memory access.

---, the target system failed to provide the debugger with any information.

**Overflow** in red, the communication channel failed to transmit all data from the target system.

### Value

Displays the access type and the value (using the access size) for the location or area you want to log accesses to. For example, if zero is read using a byte access it will be displayed as 0x00, and for a long access it will be displayed as 0x00000000.

To specify what data you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 115.

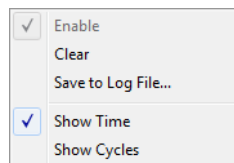
### Address

The actual memory address that is accessed. For example, if only a byte of a word is accessed, only the address of the byte is displayed. The address is calculated as base address + offset, where the base address is retrieved from the **Data Log** breakpoint dialog box and the offset is retrieved from the logs. If the log from the target system does not provide the debugger with an offset, the offset contains + ?.

\* You can double-click a line in the display area. If the value of the PC for that line is available in the source code, the editor window displays the corresponding source code (this does not include library source code).

### Context menu

This context menu is available in the **Data Log** window and the **Data Log Summary** window:



**Note:** The commands are the same in each window, but they only operate on the specific window.

These commands are available:

### Enable

Enables the logging system. The system will log information also when the window is closed.

### Clear

Deletes the log information. Note that this will happen also when you reset the debugger.

### Save to log file

Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF`. An **X** in the **Approx** column indicates that the timestamp is an approximation.

### Show Time

Displays the **Time** column in the **Data Log** window.

This menu command might not be available in the C-SPY driver you are using, which means that the **Time** column is by default displayed in the **Data Log** window.

### Show Cycles

Displays the **Cycles** column in the **Data Log** window.

This menu command might not be available in the C-SPY driver you are using, which means that the **Cycles** column is not supported.

## Data Log Summary window

The **Data Log Summary** window is available from the C-SPY driver menu.

Data	Total Accesses	Read Accesses	Write Accesses	Unknown Accesses
tVar1	42	0	25	17
tVar2	66	17	49	0
tVar3	32	32	0	0

Approximative time count: 16  
 Overflow count: 8  
 Current time: 4301.52 us

This window displays a summary of data accesses to specific memory location or areas.

See also *Getting started using data logging*, page 92.

## Requirements

The C-SPY simulator.

## Display area

Each row in this area displays the type and the number of accesses to each memory location or area in these columns:

### Data

The name of the data object you have selected to log accesses to. To specify what data object you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 115.

The current time or cycles is displayed—execution time since the start of execution or the number of cycles. Overflow count displays the number of overflows.

### Total Accesses

The number of total accesses.

If the sum of read accesses and write accesses is less than the total accesses, there have been a number of access logs for which the target system for some reason did not provide valid access type information.

### Read Accesses

The number of total read accesses.

### Write Accesses

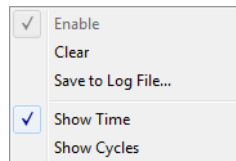
The number of total write accesses.

### Unknown Accesses

The number of unknown accesses, in other words, accesses where the access type is not known.

## Context menu

This context menu is available in the **Data Log** window and the **Data Log Summary** window:



**Note:** The commands are the same in each window, but they only operate on the specific window.

These commands are available:

**Enable**

Enables the logging system. The system will log information also when the window is closed.

**Clear**

Deletes the log information. Note that this will happen also when you reset the debugger.

**Save to log file**

Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF`. An **X** in the **Approx** column indicates that the timestamp is an approximation.

**Show Time**

Displays the **Time** column in the **Data Log** window.

This menu command might not be available in the C-SPY driver you are using, which means that the **Time** column is by default displayed in the **Data Log** window.

**Show Cycles**

Displays the **Cycles** column in the **Data Log** window.

This menu command might not be available in the C-SPY driver you are using, which means that the **Cycles** column is not supported.





# Breakpoints

- Introduction to setting and using breakpoints
- Setting breakpoints
- Reference information on breakpoints

---

## Introduction to setting and using breakpoints

These topics are covered:

- Reasons for using breakpoints
- Briefly about setting breakpoints
- Breakpoint types
- Breakpoint icons
- Breakpoints in the C-SPY simulator
- Breakpoints in the C-SPY hardware debugger drivers
- Breakpoint consumers

### REASONS FOR USING BREAKPOINTS

C-SPY® lets you set various types of breakpoints in the application you are debugging, allowing you to stop at locations of particular interest. You can set a breakpoint at a *code* location to investigate whether your program logic is correct, or to get trace printouts. In addition to code breakpoints, and depending on what C-SPY driver you are using, additional breakpoint types might be available. For example, you might be able to set a *data* breakpoint, to investigate how and when the data changes.

You can let the execution stop under certain *conditions*, which you specify. You can also let the breakpoint trigger a *side effect*, for instance executing a C-SPY macro function, by transparently stopping the execution and then resuming. The macro function can be defined to perform a wide variety of actions, for instance, simulating hardware behavior.

All these possibilities provide you with a flexible tool for investigating the status of your application.

### BRIEFLY ABOUT SETTING BREAKPOINTS

You can set breakpoints in many various ways, allowing for different levels of interaction, precision, timing, and automation. All the breakpoints you define will

appear in the Breakpoints window. From this window you can conveniently view all breakpoints, enable and disable breakpoints, and open a dialog box for defining new breakpoints. The **Breakpoint Usage** window also lists all internally used breakpoints, see *Breakpoint consumers*, page 118.

Breakpoints are set with a higher precision than single lines, using the same mechanism as when stepping; for more information about the precision, see *Single stepping*, page 70.

You can set breakpoints while you edit your code even if no debug session is active. The breakpoints will then be validated when the debug session starts. Breakpoints are preserved between debug sessions.

**Note:** For most hardware debugger systems it is only possible to set breakpoints when the application is not executing.

## BREAKPOINT TYPES

Depending on the C-SPY driver you are using, C-SPY supports different types of breakpoints.

### Code breakpoints

Code breakpoints are used for code locations to investigate whether your program logic is correct or to get trace printouts. Code breakpoints are triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will stop, before the instruction is executed.

### Log breakpoints

Log breakpoints provide a convenient way to add trace printouts without having to add any code to your application source code. Log breakpoints are triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily stop and print the specified message in the C-SPY **Debug Log** window.

### Trace Start and Stop breakpoints

Trace Start and Stop breakpoints start and stop trace data collection—a convenient way to analyze instructions between two execution points.

### Data breakpoints

Data breakpoints are primarily useful for variables that have a fixed address in memory. If you set a breakpoint on an accessible local variable, the breakpoint is set on the corresponding memory location. The validity of this location is only guaranteed for

small parts of the code. Data breakpoints are triggered when data is accessed at the specified location. The execution will usually stop directly after the instruction that accessed the data has been executed.

### Data Log breakpoints

Data log breakpoints are triggered when a specified memory address is accessed. A log entry is written in the **Data Log** window for each access. Data logs can also be displayed on the Data Log graph in the **Timeline** window, if that window is enabled.

You can set data log breakpoints using the **Breakpoints** window, the **Memory** window, and the editor window.

Using a single instruction, the microcontroller can only access values that are one byte. If you specify a data log breakpoint on a memory location that cannot be accessed by one instruction, for example a `double` or a too large area in the **Memory** window, the result might not be what you intended.

### Immediate breakpoints

The C-SPY Simulator lets you set *immediate* breakpoints, which will halt instruction execution only temporarily. This allows a C-SPY macro function to be called when the simulated processor is about to read data from a location or immediately after it has written data. Instruction execution will resume after the action.

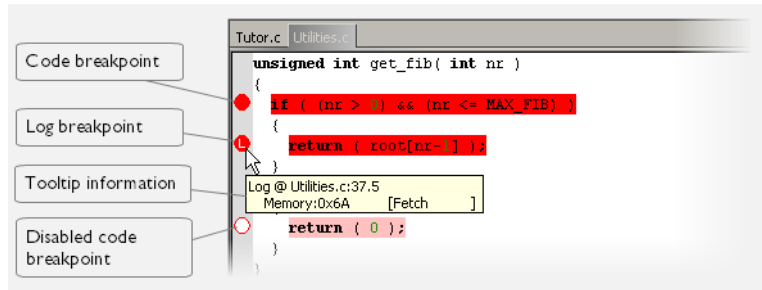
This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the simulated processor reads from a memory-mapped location, a C-SPY macro function can intervene and supply appropriate data. Conversely, when the simulated processor writes to a memory-mapped location, a C-SPY macro function can act on the value that was written.

### Complex breakpoints

The C-SPY Atmel-ICE driver, the C-SPY AVR ONE! driver, and the C-SPY JTAGICE3 driver support complex breakpoints. Complex breakpoints use the functionality of the firmware and are faster than data breakpoints and code breakpoints. Using complex breakpoints, you can specify special conditions for when the breakpoint should trigger.

## BREAKPOINT ICONS

A breakpoint is marked with an icon in the left margin of the editor window, and the icon varies with the type of breakpoint:



If the breakpoint icon does not appear, make sure the option **Show bookmarks** is selected, see Editor options in the *IDE Project Management and Building Guide*.



Just point at the breakpoint icon with the mouse pointer to get detailed tooltip information about all breakpoints set on the same location. The first row gives user breakpoint information, the following rows describe the physical breakpoints used for implementing the user breakpoint. The latter information can also be seen in the **Breakpoint Usage** window.

**Note:** The breakpoint icons might look different for the C-SPY driver you are using.

## BREAKPOINTS IN THE C-SPY SIMULATOR

The C-SPY simulator supports all breakpoint types and you can set an unlimited amount of breakpoints.

## BREAKPOINTS IN THE C-SPY HARDWARE DEBUGGER DRIVERS

Using the C-SPY drivers for hardware debugger systems you can set various breakpoint types. The amount of breakpoints you can set depends on the number of *hardware breakpoints* available on the target system or whether you have enabled *software breakpoints*, in which case the number of breakpoints you can set is unlimited.

This table summarizes the characteristics of breakpoints for the different target systems:

<b>C-SPY hardware debugger driver</b>	<b>Code breakpoints</b>	<b>Data breakpoints</b>
JTAGICE		
using hardware breakpoints <sup>8</sup>	4 <sup>1</sup>	2 <sup>1</sup>
using software breakpoints	Unlimited	2 <sup>1</sup>
JTAGICE mkII		
using hardware breakpoints <sup>3,8</sup>	4 <sup>1,2</sup>	2 <sup>1,4</sup>
using software breakpoints	Unlimited	2 <sup>1,4</sup>
Atmel-ICE		
using hardware breakpoints <sup>3,5,7, 8</sup>	4 <sup>1,2</sup>	2 <sup>6</sup>
using software breakpoints	Unlimited	2 <sup>6</sup>
JTAGICE3		
using hardware breakpoints <sup>3,5,7, 8</sup>	4 <sup>1,2</sup>	2 <sup>6</sup>
using software breakpoints	Unlimited	2 <sup>6</sup>
AVR ONE!		
using hardware breakpoints <sup>3,5,7, 8</sup>	4 <sup>1,2</sup>	2 <sup>6</sup>
using software breakpoints	Unlimited	2 <sup>6</sup>
ICE200	Unlimited	None
CCR	Unlimited	None

*Table 6: Available breakpoints in C-SPY hardware debugger drivers*

<sup>1</sup> The sum of code and data breakpoints can never exceed 4—the number of available hardware breakpoints. This means that for every data breakpoint in use, one less code breakpoint is available, and that no data breakpoints are available if you use four code breakpoints.

<sup>2</sup> If software breakpoints are enabled, the number of code breakpoints is unlimited.

<sup>3</sup> When the number of available hardware breakpoints is exceeded, software breakpoints will be used if enabled.

<sup>4</sup> Data breakpoints are not available when the debugWIRE interface is used.

<sup>5</sup> If data breakpoints and complex breakpoints have not been used, hardware breakpoints will be used until exhausted. After that, software breakpoints will be used.

<sup>6</sup> If complex breakpoints are used, data breakpoints are not available, and vice versa.

<sup>7</sup> Note that a complex breakpoint uses all available hardware breakpoints.

<sup>8</sup> The number of available hardware breakpoints depends on the target system you are using.

For Atmel-ICE, JTAGICE3, JTAGICE mkII, AVR ONE!, and Dragon, the number and types of breakpoints available depend on whether the device is using the JTAG or the debugWIRE interface. The information in this guide reflects the JTAG interface. When a device with debugWIRE is used, data breakpoints are not available and the debugger will use software code breakpoints.

If the driver and the device support software breakpoints and they are enabled, the debugger will first use any available hardware breakpoints before using software breakpoints. Exceeding the number of available hardware breakpoints, when software breakpoints are not enabled, causes the debugger to single step. This will significantly reduce the execution speed. For this reason you must be aware of the different breakpoint consumers.

## BREAKPOINT CONSUMERS

A debugger system includes several consumers of breakpoints.

### User breakpoints

The breakpoints you define in the breakpoint dialog box or by toggling breakpoints in the editor window often consume one physical breakpoint each, but this can vary greatly. Some user breakpoints consume several physical breakpoints and conversely, several user breakpoints can share one physical breakpoint. User breakpoints are displayed in the same way both in the **Breakpoint Usage** window and in the **Breakpoints** window, for example **Data @[R] callCount**.

### C-SPY itself

C-SPY itself also consumes breakpoints. C-SPY will set a breakpoint if:

- The debugger option **Run to** has been selected, and any step command is used. These are temporary breakpoints which are only set during a debug session. This means that they are not visible in the Breakpoints window.
- The linker option **With I/O emulation modules** has been selected.

In the DLIB runtime environment, C-SPY will set a system breakpoint on the `__DebugBreak` label.

In the CLIB runtime environment, C-SPY will set a breakpoint if:

- the library functions `putchar` and `getchar` are used (low-level routines used by functions like `printf` and `scanf`)
- the application has an `exit` label.

You can disable the setting of system breakpoints on the `putchar` and `getchar` functions and on the `exit` label; see .

For more information about the option **System breakpoints on**:

- For AVRONE!, see *AVR ONE! 2*, page 301.
- For JTAGICE, see *JTAGICE 2*, page 311.
- For Atmel-ICE, see *Atmel-ICE 2*, page 314.
- For JTAGICE3, see *JTAGICE3 2*, page 318.
- For JTAGICE mkII, see *JTAGICE mkII 2*, page 322.
- For Dragon, see *Dragon 2*, page 325.

These types of breakpoint consumers are displayed in the **Breakpoint Usage** window, for example, **C-SPY Terminal I/O & libsupport module**.

### **C-SPY plugin modules**

For example, modules for real-time operating systems can consume additional breakpoints. Specifically, by default, the **Stack** window consumes one physical breakpoint.

**To disable the breakpoint used by the Stack window:**

- 1** Choose **Tools>Options>Stack**.
- 2** Deselect the **Stack pointer(s) not valid until program reaches: *label*** option.

To disable the **Stack** window entirely, choose **Tools>Options>Stack** and make sure all options are deselected.

---

## **Setting breakpoints**

These tasks are covered:

- Various ways to set a breakpoint
- Toggling a simple code breakpoint
- Setting breakpoints using the dialog box
- Setting a data breakpoint in the Memory window
- Setting breakpoints using system macros
- Useful breakpoint hints.

### **VARIOUS WAYS TO SET A BREAKPOINT**

You can set a breakpoint in various ways:

- Toggling a simple code breakpoint.

- Using the **New Breakpoints** dialog box and the **Edit Breakpoints** dialog box available from the context menus in the editor window, **Breakpoints** window, and in the **Disassembly** window. The dialog boxes give you access to all breakpoint options.
- Setting a data breakpoint on a memory area directly in the **Memory** window.
- Using predefined system macros for setting breakpoints, which allows automation.

The different methods offer different levels of simplicity, complexity, and automation.

### TOGGLING A SIMPLE CODE BREAKPOINT

Toggleing a code breakpoint is a quick method of setting a breakpoint. The following methods are available both in the editor window and in the **Disassembly** window:



- Click in the gray left-side margin of the window
- Place the insertion point in the C source statement or assembler instruction where you want the breakpoint, and click the **Toggle Breakpoint** button in the toolbar
- Choose **Edit>Toggle Breakpoint**
- Right-click and choose **Toggle Breakpoint** from the context menu.

### SETTING BREAKPOINTS USING THE DIALOG BOX

The advantage of using a breakpoint dialog box is that it provides you with a graphical interface where you can interactively fine-tune the characteristics of the breakpoints. You can set the options and quickly test whether the breakpoint works according to your intentions.

All breakpoints you define using a breakpoint dialog box are preserved between debug sessions.

You can open the dialog box from the context menu available in the editor window, **Breakpoints** window, and in the **Disassembly** window.

#### To set a new breakpoint:

- 1 Choose **View>Breakpoints** to open the **Breakpoints** window.
- 2 In the **Breakpoints** window, right-click, and choose **New Breakpoint** from the context menu.
- 3 On the submenu, choose the breakpoint type you want to set.

Depending on the C-SPY driver you are using, different breakpoint types are available.

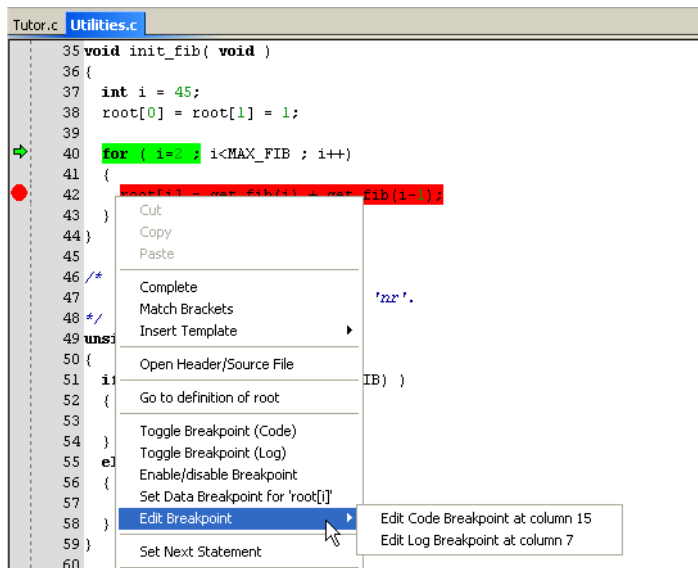
- 4 In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.

The breakpoint is displayed in the **Breakpoints** window.



### To modify an existing breakpoint:

- 1 In the **Breakpoints** window, editor window, or in the **Disassembly** window, select the breakpoint you want to modify and right-click to open the context menu.



If there are several breakpoints on the same source code line, the breakpoints will be listed on a submenu.

- 2 On the context menu, choose the appropriate command.
- 3 In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.

The breakpoint is displayed in the **Breakpoints** window.

### SETTING A DATA BREAKPOINT IN THE MEMORY WINDOW

You can set breakpoints directly on a memory location in the **Memory** window. Right-click in the window and choose the breakpoint command from the context menu that appears. To set the breakpoint on a range, select a portion of the memory contents.

The breakpoint is not highlighted in the **Memory** window; instead, you can see, edit, and remove it using the **Breakpoints** window, which is available from the **View** menu. The breakpoints you set in the **Memory** window will be triggered for both read and write accesses. All breakpoints defined in this window are preserved between debug sessions.

**Note:** Setting breakpoints directly in the **Memory** window is only possible if the driver you use supports this.

## SETTING BREAKPOINTS USING SYSTEM MACROS

You can set breakpoints not only in the breakpoint dialog box but also by using built-in C-SPY system macros. When you use system macros for setting breakpoints, the breakpoint characteristics are specified as macro parameters.

Macros are useful when you have already specified your breakpoints so that they fully meet your requirements. You can define your breakpoints in a macro file, using built-in system macros, and execute the file at C-SPY startup. The breakpoints will then be set automatically each time you start C-SPY. Another advantage is that the debug session will be documented, and that several engineers involved in the development project can share the macro files.

**Note:** If you use system macros for setting breakpoints, you can still view and modify them in the Breakpoints window. In contrast to using the dialog box for defining breakpoints, all breakpoints that are defined using system macros are removed when you exit the debug session.

These breakpoint macros are available:

C-SPY macro for breakpoints	Sim	Atmel-ICE	JTAGICE3	JTAGICE mkII	AVR ONE!	JTAGICE	ICE200	CCR
__setCodeBreak	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
__setDataBreak	Yes	Yes	Yes	—	Yes	—	—	—
__setLogBreak	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
__setDataLogBreak	Yes	—	—	—	—	—	—	—
__setSimBreak	Yes	—	—	—	—	—	—	—
__setTraceStartBreak	Yes	—	—	—	—	—	—	—
__setTraceStopBreak	Yes	—	—	—	—	—	—	—
__clearBreak	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 7: C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 221.

## Setting breakpoints at C-SPY startup using a setup macro file

You can use a setup macro file to define breakpoints at C-SPY startup. Follow the procedure described in *Using C-SPY macros*, page 209.

## USEFUL BREAKPOINT HINTS

Below are some useful hints related to setting breakpoints.



### Tracing incorrect function arguments

If a function with a pointer argument is sometimes incorrectly called with a `NULL` argument, you might want to debug that behavior. These methods can be useful:

- Set a breakpoint on the first line of the function with a condition that is true only when the parameter is 0. The breakpoint will then not be triggered until the problematic situation actually occurs. The advantage of this method is that no extra source code is needed. The drawback is that the execution speed might become unacceptably low.
- You can use the `assert` macro in your problematic function, for example:

```
int MyFunction(int * MyPtr)
{
    assert(MyPtr != 0); /* Assert macro added to your source
                        code. */
    /* Here comes the rest of your function. */
}
```

The execution will break whenever the condition is true. The advantage is that the execution speed is only very slightly affected, but the drawback is that you will get a small extra footprint in your source code. In addition, the only way to get rid of the execution stop is to remove the macro and rebuild your source code.

- Instead of using the `assert` macro, you can modify your function like this:

```
int MyFunction(int * MyPtr)
{
    if(MyPtr == 0)
        MyDummyStatement; /* Dummy statement where you set a
                            breakpoint. */
    /* Here comes the rest of your function. */
}
```

You must also set a breakpoint on the extra dummy statement, so that the execution will break whenever the condition is true. The advantage is that the execution speed is only very slightly affected, but the drawback is that you will still get a small extra footprint in your source code. However, in this way you can get rid of the execution stop by just removing the breakpoint.



### Performing a task and continuing execution

You can perform a task when a breakpoint is triggered and then automatically continue execution.

You can use the **Action** text box to associate an action with the breakpoint, for instance a C-SPY macro function. When the breakpoint is triggered and the execution of your application has stopped, the macro function will be executed. In this case, the execution will not continue automatically.

Instead, you can set a condition which returns 0 (false). When the breakpoint is triggered, the condition—which can be a call to a C-SPY macro that performs a task—is evaluated and because it is not true, execution continues.

Consider this example where the C-SPY macro function performs a simple task:

```
__var my_counter;

count ()
{
    my_counter += 1;
    return 0;
}
```

To use this function as a condition for the breakpoint, type `count ()` in the **Expression** text box under **Conditions**. The task will then be performed when the breakpoint is triggered. Because the macro function `count` returns 0, the condition is false and the execution of the program will resume automatically, without any stop.

---

## Reference information on breakpoints

Reference information about:

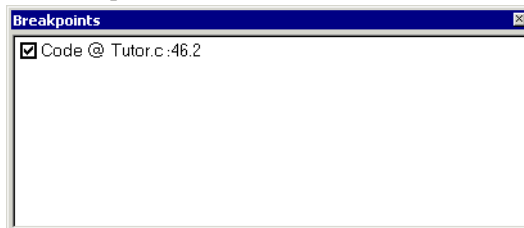
- *Breakpoints window*, page 125
- *Breakpoint Usage window*, page 126
- *Code breakpoints dialog box*, page 127
- *Log breakpoints dialog box*, page 129
- *Data breakpoints dialog box*, page 130
- *Data Log breakpoints dialog box*, page 132
- *Immediate breakpoints dialog box*, page 133
- *Complex breakpoints dialog box*, page 135
- *Enter Location dialog box*, page 138
- *Resolve Source Ambiguity dialog box*, page 139.

See also:

- *Reference information on C-SPY system macros*, page 221
- *Reference information on trace*, page 168.

## Breakpoints window

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



This window lists all breakpoints you define.

Use this window to conveniently monitor, enable, and disable breakpoints; you can also define new breakpoints and modify existing breakpoints.

### Requirements

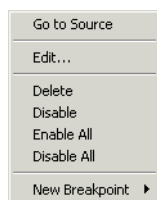
None; this window is always available.

### Display area

This area lists all breakpoints you define. For each breakpoint, information about the breakpoint type, source file, source line, and source column is provided.

### Context menu

This context menu is available:



These commands are available:

#### Go to Source

Moves the insertion point to the location of the breakpoint, if the breakpoint has a source location. Double-click a breakpoint in the **Breakpoints** window to perform the same command.

#### Edit

Opens the breakpoint dialog box for the breakpoint you selected.

**Delete**

Deletes the breakpoint. Press the Delete key to perform the same command.

**Enable**

Enables the breakpoint. The check box at the beginning of the line will be selected. You can also perform the command by manually selecting the check box. This command is only available if the breakpoint is disabled.

**Disable**

Disables the breakpoint. The check box at the beginning of the line will be deselected. You can also perform this command by manually deselecting the check box. This command is only available if the breakpoint is enabled.

**Enable All**

Enables all defined breakpoints.

**Disable All**

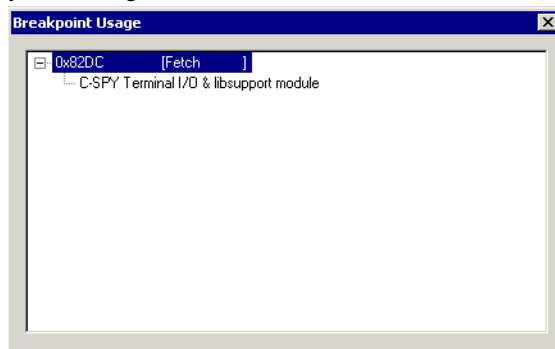
Disables all defined breakpoints.

**New Breakpoint**

Displays a submenu where you can open the breakpoint dialog box for the available breakpoint types. All breakpoints you define using this dialog box are preserved between debug sessions.

## Breakpoint Usage window

The **Breakpoint Usage** window is available from the menu specific to the C-SPY driver you are using.



This window lists all breakpoints currently set in the target system, both the ones you have defined and the ones used internally by C-SPY. The format of the items in this window depends on the C-SPY driver you are using.

The window gives a low-level view of all breakpoints, related but not identical to the list of breakpoints displayed in the **Breakpoints** window.

C-SPY uses breakpoints when stepping. see the **Breakpoint Usage** window for:

- Identifying all breakpoint consumers
- Checking that the number of active breakpoints is supported by the target system
- Configuring the debugger to use the available breakpoints in a better way, if possible.

For more information, see .

### Requirements

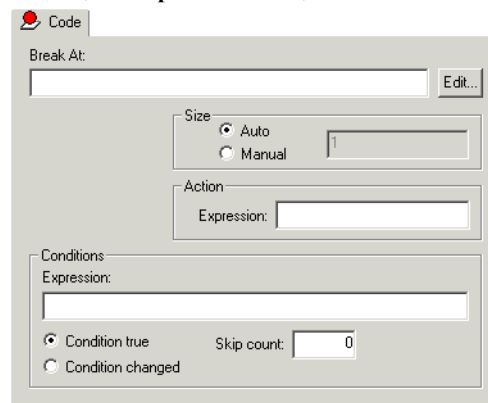
None; this window is always available.

### Display area

For each breakpoint in the list, the address and access type are displayed. Each breakpoint in the list can also be expanded to show its originator.

## Code breakpoints dialog box

The **Code** breakpoints dialog box is available from the context menu in the editor window, **Breakpoints** window, and in the **Disassembly** window.



This figure reflects the C-SPY simulator.

Use the **Code** breakpoints dialog box to set a code breakpoint.

### Requirements

None; this dialog box is always available.

### **Break At**

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 138.

### **Size**

Determines whether there should be a size—in practice, a range—of locations where the breakpoint will trigger. Each fetch access to the specified memory range will trigger the breakpoint. Select how to specify the size:

#### **Auto**

The size will be set automatically, typically to 1.

#### **Manual**

Specify the size of the breakpoint range in the text box.

### **Action**

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 123.

### **Conditions**

Specify simple or complex conditions:

#### **Expression**

Specify a valid C-SPY expression, see *C-SPY expressions*, page 88.

#### **Condition true**

The breakpoint is triggered if the value of the expression is true.

#### **Condition changed**

The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

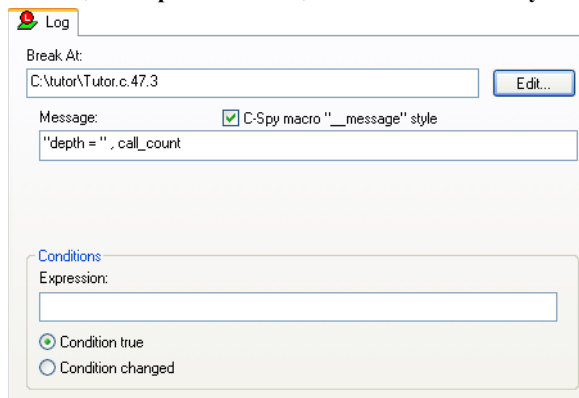
#### **Skip count**

The number of times that the breakpoint condition must be fulfilled before the breakpoint starts triggering. After that, the breakpoint will trigger every time the condition is fulfilled.



## Log breakpoints dialog box

The **Log** breakpoints dialog box is available from the context menu in the editor window, **Breakpoints** window, and in the **Disassembly** window.



This figure reflects the C-SPY simulator.

Use the **Log** breakpoints dialog box to set a log breakpoint.

### Requirements

None; this dialog box is always available.

### Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 138.

### Message

Specify the message you want to be displayed in the C-SPY **Debug Log** window. The message can either be plain text, or—if you also select the option **C-SPY macro \"\_\_message\" style**—a comma-separated list of arguments.

### C-SPY macro \"\_\_message\" style

Select this option to make a comma-separated list of arguments specified in the **Message** text box be treated exactly as the arguments to the C-SPY macro language statement `__message`, see *Formatted output*, page 217.

## Conditions

Specify simple or complex conditions:

### Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 88.

### Condition true

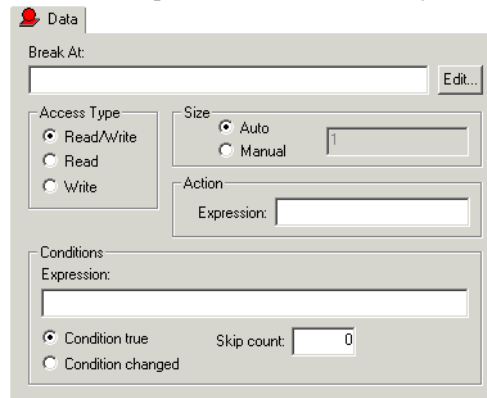
The breakpoint is triggered if the value of the expression is true.

### Condition changed

The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

## Data breakpoints dialog box

The **Data** breakpoints dialog box is available from the context menu in the editor window, **Breakpoints** window, the **Memory** window, and in the **Disassembly** window.



This figure reflects the C-SPY simulator.

Use the **Data** breakpoints dialog box to set a data breakpoint. Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed.

## Requirements

One of these alternatives:

- The C-SPY simulator
- The C-SPY Atmel ICE driver
- The C-SPY JTAGICE3 driver

- The C-SPY AVR ONE! driver
- The C-SPY JTAGICE driver
- The C-SPY JTAGICE mkII driver, unless the debugWIRE interface is used
- The C-SPY Dragon driver.

### Break At

Specify the data location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 138.

### Access Type

Selects the type of memory access that triggers the breakpoint:

#### Read/Write

Reads from or writes to location.

#### Read

Reads from location.

#### Write

Writes to location.

### Size

Determines whether there should be a size—in practice, a range—of locations where the breakpoint will trigger. Each fetch access to the specified memory range will trigger the breakpoint. Select how to specify the size:

#### Auto

The size will automatically be based on the type of expression the breakpoint is set on. For example, if you set the breakpoint on a 12-byte structure, the size of the breakpoint will be 12 bytes.

#### Manual

Specify the size of the breakpoint range in the text box.

For data breakpoints, this can be useful if you want the breakpoint to be triggered on accesses to data structures, such as arrays, structs, and unions.

### Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 123.

## Conditions

Specify simple or complex conditions:

### Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 88.

### Condition true

The breakpoint is triggered if the value of the expression is true.

### Condition changed

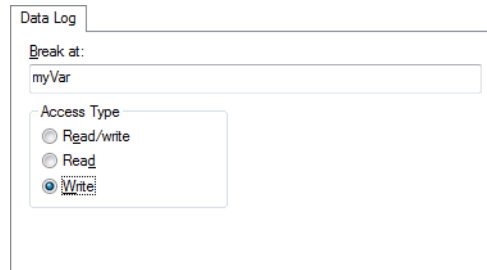
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

### Skip count

The number of times that the breakpoint condition must be fulfilled before the breakpoint starts triggering. After that, the breakpoint will trigger every time the condition is fulfilled.

## Data Log breakpoints dialog box

The **Data Log** breakpoints dialog box is available from the context menu in the **Breakpoints** window.



This figure reflects the C-SPY simulator.

Use the **Data Log** breakpoints dialog box to set a maximum of four data log breakpoints on memory addresses.

See also *Data Log breakpoints*, page 115 and *Getting started using data logging*, page 92.

## Requirements

The C-SPY simulator.

**Trigger at**

Specify the data location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 138.

**Access Type**

Selects the type of memory access that triggers the breakpoint:

**Read/Write**

Reads from or writes to location.

**Read**

Reads from location.

**Write**

Writes to location.

**Immediate breakpoints dialog box**

The **Immediate** breakpoints dialog box is available from the context menu in the editor window, **Breakpoints** window, the **Memory** window, and in the **Disassembly** window.

The screenshot shows a dialog box titled "Immediate". It has a "Trigger at:" label above a text input field, with an "Edit..." button to the right. Below the "Trigger at:" field is an "Access Type" section containing two radio buttons: "Read" (which is selected) and "Write". To the right of the "Access Type" section is an "Action" section containing an "Expression:" label and a text input field.

In the C-SPY simulator, use the **Immediate** breakpoints dialog box to set an immediate breakpoint. Immediate breakpoints do not stop execution at all; they only suspend it temporarily.

**Requirements**

The C-SPY simulator.

**Trigger at**

Specify the data location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 138.

**Access Type**

Selects the type of memory access that triggers the breakpoint:

**Read**

Reads from location.

**Write**

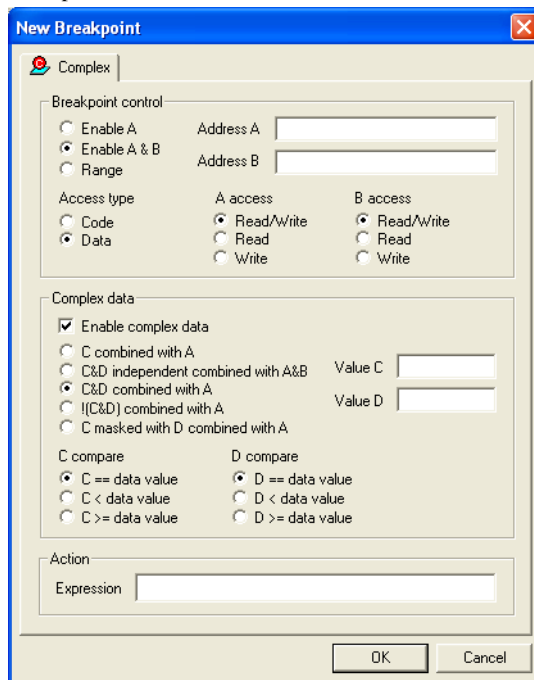
Writes to location.

**Action**

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 123.

## Complex breakpoints dialog box

The **Complex** breakpoints dialog box is available from the context menu in the Breakpoints window.



Use this dialog box to set a complex breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the Breakpoint window and choose **Edit** on the context menu.

Complex breakpoints use the functionality of the firmware and are faster than data breakpoints and code breakpoints.

**Note:** A complex breakpoint uses all available hardware breakpoints.

### Requirements

One of these alternatives:

- The C-SPY Atmel-ICE driver
- The C-SPY JTAGICE3 driver
- The C-SPY AVR ONE! driver.

**Breakpoint control**

Controls what access type at the specified address that causes a break. Choose between:

<b>Enable A</b>	Breaks at the address specified in the <b>Address A</b> text box.
<b>Enable A&amp;B</b>	Breaks both at the address specified in <b>Address A</b> and at the address in <b>Address B</b> .
<b>Range</b>	Breaks when an address from <b>Address A</b> up to and including <b>Address B</b> is accessed. This can be useful if you want the breakpoint to be triggered on access to data structures, such as arrays, structs, and unions. When using <b>Range</b> , only <b>A access</b> is available.
<b>A access/B access</b>	Specifies the type of memory access that triggers complex breakpoints.  <b>Read/Write</b> , Reads from or writes to location <b>Read</b> , Reads from location <b>Write</b> , Writes to location

**Address A/B**

Specify the code or data addresses where you want to set a breakpoint.

**Access type**

Selects the memory space, **Code** or **Data**, for the addresses in **Address A** and **Address B**. Note that both addresses must have the same access type.

**Complex data**

**Enable complex data** enables the data compare functionality.

**Value C/D**

Specify 1-byte numbers for the compare functionality.

<b>C combined with A</b>	Breaks when <b>Address A</b> is accessed using <b>A access</b> and <b>Value C</b> matches the memory contents at <b>Address A</b> according to <b>C compare</b> .
--------------------------	---



<b>C&amp;D independent combined with A&amp;B</b>	<p>Breaks when <b>Address A</b> is accessed using <b>A access</b> and <b>Value C</b> matches the memory contents at <b>Address A</b> according to <b>C compare</b></p> <p><i>or</i></p> <p>when <b>Address B</b> is accessed using <b>B access</b> and <b>Value D</b> matches the memory contents at <b>Address B</b> according to <b>D compare</b>.</p>
<b>C&amp;D combined with A</b>	Breaks when <b>Address A</b> is accessed using <b>A access</b> and <b>Value C</b> matches the memory contents at <b>Address A</b> according to <b>C compare</b> and <b>Value D</b> matches the memory contents at <b>Address A</b> according to <b>D compare</b> .
<b>(C&amp;D) combined with A</b>	Breaks when <b>Address A</b> is accessed using <b>A access</b> and <b>Value C</b> does not match the memory contents of <b>Address A</b> according to <b>C compare</b> and/or <b>Value D</b> does not match the memory contents of <b>Address A</b> according to <b>D compare</b> .
<b>C masked with D combined with A</b>	Breaks when <b>Address A</b> is accessed using <b>A access</b> and <b>Value C</b> masked with <b>Value D</b> matches the memory contents at <b>Address A</b> according to <b>C compare</b> .

**C/D compare**

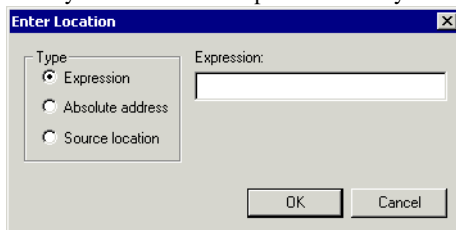
Specify the relationship between **Value C** or **Value D** and the contents of data memory at **Address A** and **Address B**.

**Action**

Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

## Enter Location dialog box

The **Enter Location** dialog box is available from the breakpoints dialog box, either when you set a new breakpoint or when you edit a breakpoint.



Use the **Enter Location** dialog box to specify the location of the breakpoint.

**Note:** This dialog box looks different depending on the **Type** you select.

### Type

Selects the type of location to be used for the breakpoint, choose between:

#### Expression

A C-SPY expression, whose value evaluates to a valid code or data location.

A code location, for example the function `main`, is typically used for code breakpoints.

A data location is the name of a variable and is typically used for data breakpoints. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the location of the fourth element of the array `arr`. For static variables declared with the same name in several functions, use the syntax `my_func::my_static_variable` to refer to a specific variable.

For more information about C-SPY expressions, see *C-SPY expressions*, page 88.

#### Absolute address

An absolute location on the form `zone:hexaddress` or simply `hexaddress` (for example `Memory:0x42`). `zone` refers to C-SPY memory zones and specifies in which memory the address belongs, see *C-SPY memory zones*, page 142.

#### Source location

A location in your C source code using the syntax:

```
{filename}.row.column.
```

`filename` specifies the filename and full path.

`row` specifies the row in which you want the breakpoint.

*column* specifies the column in which you want the breakpoint.

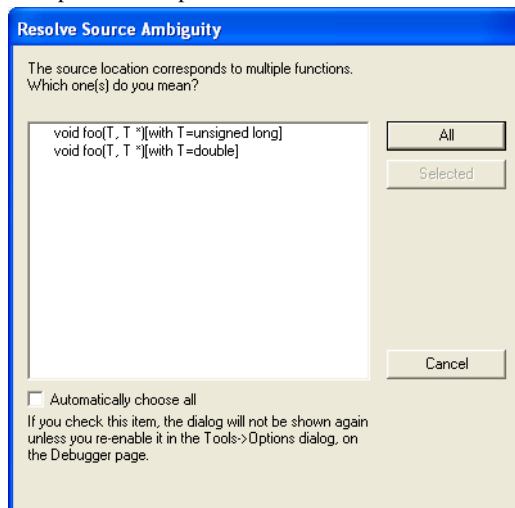
For example, `{C:\src\prog.c}.22.3`

sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that in quoted form, for example in a C-SPY macro, you must instead write `{C:\src\prog.c}.22.3`.

Note that the Source location type is usually meaningful only for code locations in code breakpoints. Depending on the C-SPY driver you are using, **Source location** might not be available for data and immediate breakpoints.

## Resolve Source Ambiguity dialog box

The **Resolve Source Ambiguity** dialog box appears, for example, when you try to set a breakpoint on templates and the source location corresponds to more than one function.



To resolve a source ambiguity, perform one of these actions:

- In the text box, select one or several of the listed locations and click **Selected**.
- Click **All**.

### All

The breakpoint will be set on all listed locations.

### Selected

The breakpoint will be set on the source locations that you have selected in the text box.

**Cancel**

No location will be used.

**Automatically choose all**

Determines that whenever a specified source location corresponds to more than one function, all locations will be used.

Note that this option can also be specified in the **IDE Options** dialog box, see Debugger options in the *IDE Project Management and Building Guide*.

# Memory and registers

- Introduction to monitoring memory and registers
- Monitoring memory and registers
- Reference information on memory and registers

---

## Introduction to monitoring memory and registers

These topics are covered:

- Briefly about monitoring memory and registers
- C-SPY memory zones
- Stack display

### BRIEFLY ABOUT MONITORING MEMORY AND REGISTERS

C-SPY provides many windows for monitoring memory and registers, each of them available from the **View** menu:

- The **Memory** window  
Gives an up-to-date display of a specified area of memory—a memory zone—and allows you to edit it. Different colors are used for indicating data coverage along with execution of your application. You can fill specified areas with specific values and you can set breakpoints directly on a memory location or range. You can open several instances of this window, to monitor different memory areas. The content of the window can be regularly updated while your application is executing.
- The **Symbolic Memory** window  
Displays how variables with static storage duration are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example by buffer overruns.
- The **Stack** window  
Displays the contents of the stack, including how stack variables are laid out in memory. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the **Stack** window is useful for determining the optimal size of the stack. You can open up to two instances of this window, each showing different stacks or different display modes of the same stack.

- The **Register** window

Gives an up-to-date display of the contents of the processor registers and SFRs, and allows you to edit them. Because of the large amount of registers—memory-mapped peripheral unit registers and CPU registers—it is inconvenient to show all registers concurrently in the **Register** window. Instead you can divide registers into *register groups*. You can choose to load either predefined register groups or define your own application-specific groups. You can open several instances of this window, each showing a different register group.

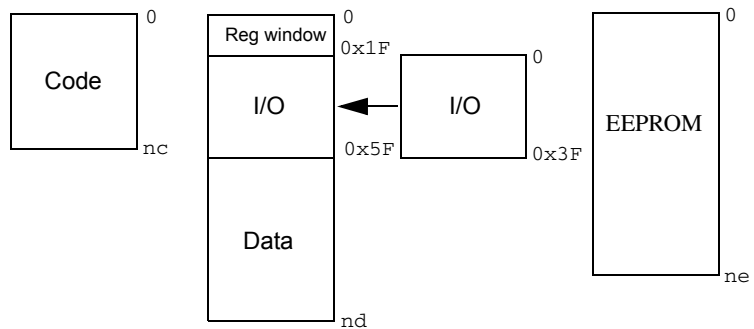
To view the memory contents for a specific variable, simply drag the variable to the **Memory** window or the **Symbolic** memory window. The memory area where the variable is located will appear.



Reading the value of some registers might influence the runtime behavior of your application. For example, reading the value of a UART status register might reset a pending bit, which leads to the lack of an interrupt that would have processed a received byte. To prevent this from happening, make sure that the Register window containing any such registers is closed when debugging a running application.

### C-SPY MEMORY ZONES

In C-SPY, the term *zone* is used for a named memory area. A memory address, or *location*, is a combination of a zone and a numerical offset into that zone. By default, four address zones—CODE, DATA, EEPROM, and IO\_SPACE—in the debugger cover the whole AVR memory range.



nc to ne indicate the size of the memories

Memory zones are used in several contexts, most importantly in the **Memory** and **Disassembly** windows, and in C-SPY macros. In the windows, use the **Zone** box to choose which memory zone to display.

## Device-specific zones

Memory information for device-specific zones is defined in the *device description files*. When you load a device description file, additional zones that adhere to the specific memory layout become available.

See the device description file for information about available memory zones.

If your hardware does not have the same memory layout as any of the predefined device description files, you can define customized zones by adding them to the file.

For more information, see *Selecting a device description file*, page 52 and *Modifying a device description file*, page 56.

## STACK DISPLAY

The **Stack** window displays the contents of the stack, overflow warnings, and it has a graphical stack bar. These can be useful in many contexts. Some examples are:

- Investigating the stack usage when assembler modules are called from C modules and vice versa
- Investigating whether the correct elements are located on the stack
- Investigating whether the stack is restored properly
- Determining the optimal stack size
- Detecting stack overflows.

For microcontrollers with multiple stacks, you can select which stack to view.

### Stack usage

When your application is first loaded, and upon each reset, the memory for the stack area is filled with the dedicated byte value `0xCD` before the application starts executing. Whenever execution stops, the stack memory is searched from the end of the stack until a byte with a value different from `0xCD` is found, which is assumed to be how far the stack has been used. Although this is a reasonably reliable way to track stack usage, there is no guarantee that a stack overflow is detected. For example, a stack *can* incorrectly grow outside its bounds, and even modify memory outside the stack area, without actually modifying any of the bytes near the stack range. Likewise, your application might modify memory within the stack area by mistake.



The **Stack** window cannot detect a stack overflow when it happens, but can only detect the signs it leaves behind. However, when the graphical stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled.

**Note:** The size and location of the stack is retrieved from the definition of the segment holding the stack, made in the linker configuration file. If you, for some reason, modify the stack initialization made in the system startup code, `cstartup`, you should also

change the segment definition in the linker configuration file accordingly; otherwise the Stack window cannot track the stack usage. For more information about this, see the *IAR C/C++ Compiler Reference Guide for AVR*.

## Monitoring memory and registers

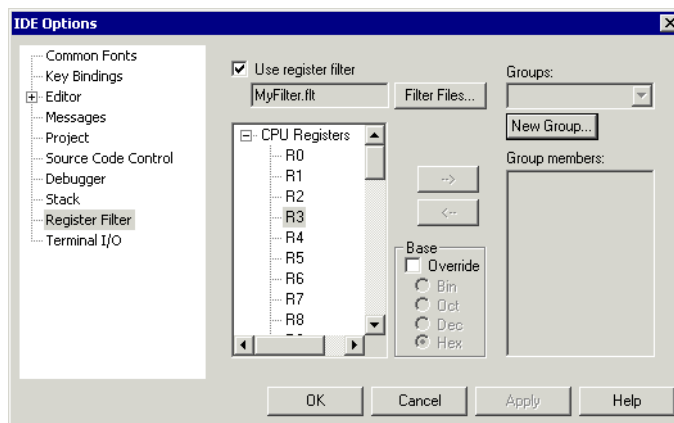
These tasks are covered:

- *Defining application-specific register groups*, page 144.

### DEFINING APPLICATION-SPECIFIC REGISTER GROUPS

Defining application-specific register groups minimizes the amount of registers displayed in the **Register** window and speeds up the debugging.

- 1 Choose **Tools>Options>Register Filter** during a debug session.

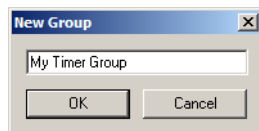


For information about the register filter options, see the *IDE Project Management and Building Guide*.

- 2 Select **Use register filter** and specify the filename and destination of the filter file for your new group in the dialog box that appears.



- 3 Click **New Group** and specify the name of your group, for example **My Timer Group**.



- 4 In the register tree view on the **Register Filter** page, select a register and click the arrow button to add it to your group. Repeat this process for all registers that you want to add to your group.
- 5 Optionally, select any registers for which you want to change the integer base, and choose a suitable base.
- 6 When you are done, click **OK**. Your new group is now available in the **Register** window.

If you want to add more groups to your filter file, repeat this procedure for each group you want to add.

**Note:** The registers that appear in the list of registers are retrieved from the `ddf` file that is currently used.

---

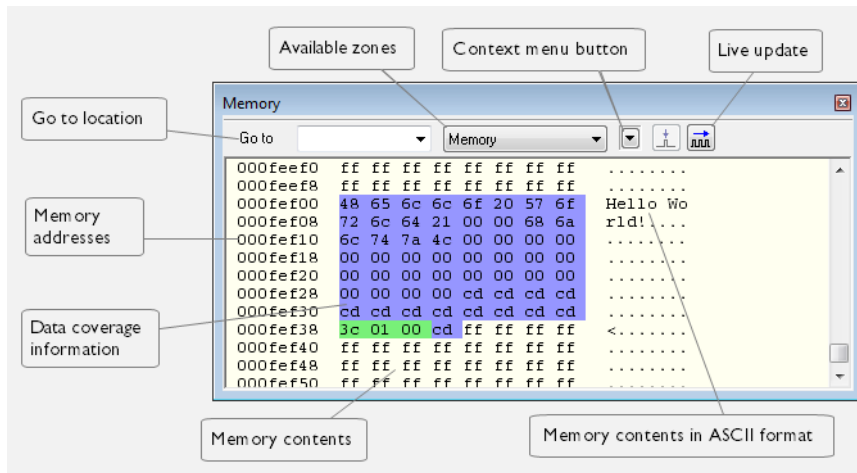
## Reference information on memory and registers

Reference information about:

- *Memory window*, page 146
- *Memory Save dialog box*, page 150
- *Memory Restore dialog box*, page 151
- *Fill dialog box*, page 152
- *Symbolic Memory window*, page 153
- *Stack window*, page 156
- *Register window*, page 159

## Memory window

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



This window gives an up-to-date display of a specified area of memory—a memory zone—and allows you to edit it. You can open several instances of this window, which is very convenient if you want to keep track of several memory or register zones, or monitor different parts of the memory.



To view the memory corresponding to a variable, you can select it in the editor window and drag it to the **Memory** window.

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 59.

### Requirements

None; this window is always available.

### Toolbar

The toolbar contains:

#### Go to

The memory location or symbol you want to view.

#### Zone

Selects a memory zone, see *C-SPY memory zones*, page 142.

**Context menu button**

Displays the context menu.

**Display area**

The display area shows the addresses currently being viewed, the memory contents in the format you have chosen, and—provided that the display mode is set to **1x Units**—the memory contents in ASCII format. You can edit the contents of the display area, both in the hexadecimal part and the ASCII part of the area.

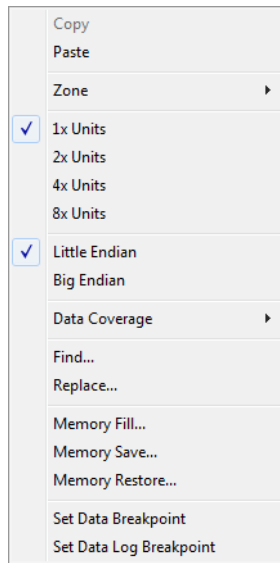
Data coverage is displayed with these colors:

Yellow	Indicates data that has been read.
Blue	Indicates data that has been written
Green	Indicates data that has been both read and written.

**Note:** Data coverage is not supported by all C-SPY drivers. Data coverage is supported by the C-SPY Simulator.

## Context menu

This context menu is available:



These commands are available:

### Copy, Paste

Standard editing commands.

### Zone

Selects a memory zone, see *C-SPY memory zones*, page 142.

### 1x Units

Displays the memory contents as single bytes.

### 2x Units

Displays the memory contents as 2-byte groups.

### 4x Units

Displays the memory contents as 4-byte groups.

### 8x Units

Displays the memory contents as 8-byte groups.

### Little Endian

Displays the contents in little-endian byte order.

**Big Endian**

Displays the contents in big-endian byte order.

**Data Coverage**

Choose between:

**Enable** toggles data coverage on or off.

**Show** toggles between showing or hiding data coverage.

**Clear** clears all data coverage information.

These commands are only available if your C-SPY driver supports data coverage.

**Find**

Displays a dialog box where you can search for text within the **Memory** window; read about the **Find** dialog box in the *IDE Project Management and Building Guide*.

**Replace**

Displays a dialog box where you can search for a specified string and replace each occurrence with another string; read about the **Replace** dialog box in the *IDE Project Management and Building Guide*.

**Memory Fill**

Displays a dialog box, where you can fill a specified area with a value, see *Fill dialog box*, page 152.

**Memory Save**

Displays a dialog box, where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 150.

**Memory Restore**

Displays a dialog box, where you can load the contents of a file in Intel-hex or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 151.

**Set Data Breakpoint**

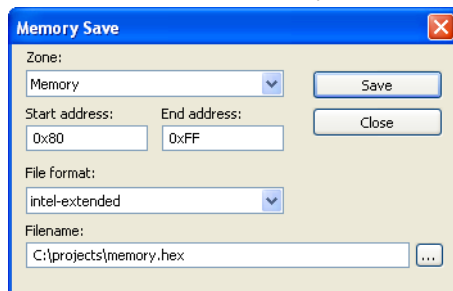
Sets breakpoints directly in the **Memory** window. The breakpoint is not highlighted; you can see, edit, and remove it in the **Breakpoints** dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the Memory window*, page 121.

### Set Data Log Breakpoint

Sets a breakpoint on the start address of a memory selection directly in the **Memory** window. The breakpoint is not highlighted; you can see, edit, and remove it in the **Breakpoints** dialog box. The breakpoints you set in this window will be triggered by both read and write accesses; to change this, use the **Breakpoints** window. For more information, see *Data Log breakpoints*, page 115 and *Getting started using data logging*, page 92.

## Memory Save dialog box

The **Memory Save** dialog box is available by choosing **Debug>Memory>Save** or from the context menu in the **Memory** window.



Use this dialog box to save the contents of a specified memory area to a file.

### Requirements

None; this dialog box is always available.

### Zone

Selects a memory zone, see *C-SPY memory zones*, page 142.

### Start address

Specify the start address of the memory range to be saved.

### End address

Specify the end address of the memory range to be saved.

### File format

Selects the file format to be used, which is Intel-extended by default.

**Filename**

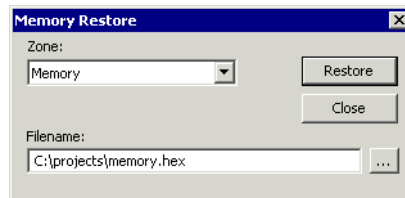
Specify the destination file to be used; a browse button is available for your convenience.

**Save**

Saves the selected range of the memory zone to the specified file.

**Memory Restore dialog box**

The **Memory Restore** dialog box is available by choosing **Debug>Memory>Restore** or from the context menu in the **Memory** window.



Use this dialog box to load the contents of a file in Intel-extended or Motorola S-record format to a specified memory zone.

**Requirements**

None; this dialog box is always available.

**Zone**

Selects a memory zone, see *C-SPY memory zones*, page 142.

**Filename**

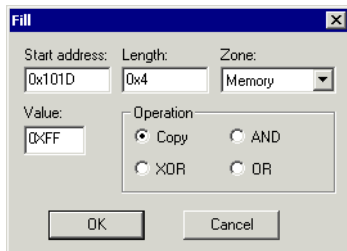
Specify the file to be read; a browse button is available for your convenience.

**Restore**

Loads the contents of the specified file to the selected memory zone.

## Fill dialog box

The **Fill** dialog box is available from the context menu in the **Memory** window.



Use this dialog box to fill a specified area of memory with a value.

### Requirements

None; this dialog box is always available.

### Start address

Type the start address—in binary, octal, decimal, or hexadecimal notation.

### Length

Type the length—in binary, octal, decimal, or hexadecimal notation.

### Zone

Selects a memory zone, see *C-SPY memory zones*, page 142.

### Value

Type the 8-bit value to be used for filling each memory location.

### Operation

These are the available memory fill operations:

#### Copy

Value will be copied to the specified memory area.

#### AND

An AND operation will be performed between Value and the existing contents of memory before writing the result to memory.

#### XOR

An XOR operation will be performed between Value and the existing contents of memory before writing the result to memory.

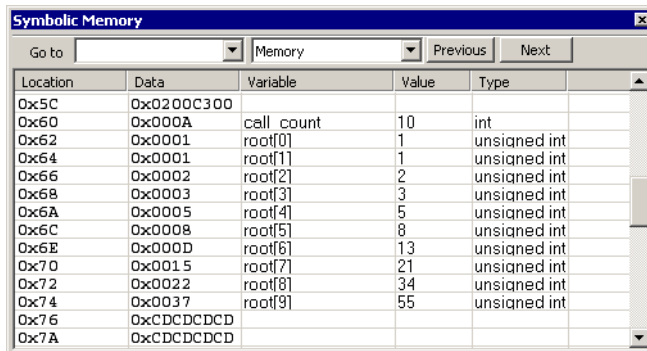


**OR**

An OR operation will be performed between Value and the existing contents of memory before writing the result to memory.

**Symbolic Memory window**

The **Symbolic Memory** window is available from the **View** menu during a debug session.



Location	Data	Variable	Value	Type
0x5C	0x0200C300			
0x60	0x000A	call count	10	int
0x62	0x0001	root[0]	1	unsigned int
0x64	0x0001	root[1]	1	unsigned int
0x66	0x0002	root[2]	2	unsigned int
0x68	0x0003	root[3]	3	unsigned int
0x6A	0x0005	root[4]	5	unsigned int
0x6C	0x0008	root[5]	8	unsigned int
0x6E	0x000D	root[6]	13	unsigned int
0x70	0x0015	root[7]	21	unsigned int
0x72	0x0022	root[8]	34	unsigned int
0x74	0x0037	root[9]	55	unsigned int
0x76	0xCDCDCDCD			
0x7A	0xCDCDCDCD			

This window displays how variables with static storage duration, typically variables with file scope but also static variables in functions and classes, are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example buffer overruns. Other areas of use are spotting alignment holes or for understanding problems caused by buffers being overwritten.



To view the memory corresponding to a variable, you can select it in the editor window and drag it to the **Symbolic Memory** window.

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 59.

**Requirements**

None; this window is always available.

**Toolbar**

The toolbar contains:

**Go to**

The memory location or symbol you want to view.

**Zone**

Selects a memory zone, see *C-SPY memory zones*, page 142.

**Previous**

Highlights the previous symbol in the display area.

**Next**

Highlights the next symbol in the display area.

**Display area**

This area contains these columns:

**Location**

The memory address.

**Data**

The memory contents in hexadecimal format. The data is grouped according to the size of the symbol. This column is editable.

**Variable**

The variable name; requires that the variable has a fixed memory location. Local variables are not displayed.

**Value**

The value of the variable. This column is editable.

**Type**

The type of the variable.

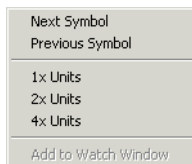
There are several different ways to navigate within the memory space:

- Text that is dropped in the window is interpreted as symbols
- The scroll bar at the right-side of the window
- The toolbar buttons **Next** and **Previous**
- The toolbar list box **Go to** can be used for locating specific locations or symbols.

**Note:** Rows are marked in red when the corresponding value has changed.

## Context menu

This context menu is available:



These commands are available:

### Next Symbol

Highlights the next symbol in the display area.

### Previous Symbol

Highlights the previous symbol in the display area.

### 1x Units

Displays the memory contents as single bytes. This applies only to rows which do not contain a variable.

### 2x Units

Displays the memory contents as 2-byte groups.

### 4x Units

Displays the memory contents as 4-byte groups.

### Add to Watch window

Adds the selected symbol to the **Watch** window.

### Default format

Displays the memory contents in the default format.

### Binary format

Displays the memory contents in binary format.

### Octal format

Displays the memory contents in octal format.

### Decimal format

Displays the memory contents in decimal format.

### Hexadecimal format

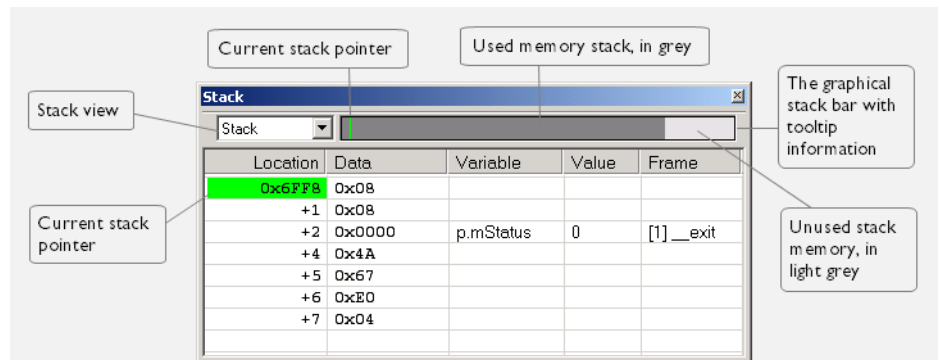
Displays the memory contents in hexadecimal format.

### Char format

Displays the memory contents in char format.

## Stack window

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



This window is a memory window that displays the contents of the stack. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the **Stack** window is useful for determining the optimal size of the stack.

### To view the graphical stack bar:

- 1 Choose **Tools>Options>Stack**.
- 2 Select the option **Enable graphical stack display and stack usage**.

You can open up to two **Stack** windows, each showing a different stack—if several stacks are available—or the same stack with different display settings.

**Note:** By default, this window uses one physical breakpoint. For more information, see *Breakpoint consumers*, page 118.

For information about options specific to the **Stack** window, see the *IDE Project Management and Building Guide*.

### Requirements

None; this window is always available.

### Toolbar

The toolbar contains:

#### Stack

Selects which stack to view. This applies to microcontrollers with multiple stacks.

## The graphical stack bar

Displays the state of the stack graphically.

The left end of the stack bar represents the bottom of the stack, in other words, the position of the stack pointer when the stack is empty. The right end represents the end of the memory space reserved for the stack. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

When the stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled.



Place the mouse pointer over the stack bar to get tooltip information about stack usage.

## Display area

This area contains these columns:

### Location

Displays the location in memory. The addresses are displayed in increasing order. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.

### Data

Displays the contents of the memory unit at the given location. From the **Stack** window context menu, you can select how the data should be displayed; as a 1-, 2-, or 4-byte group of data.

### Variable

Displays the name of a variable, if there is a local variable at the given location. Variables are only displayed if they are declared locally in a function, and located on the stack and not in registers.

### Value

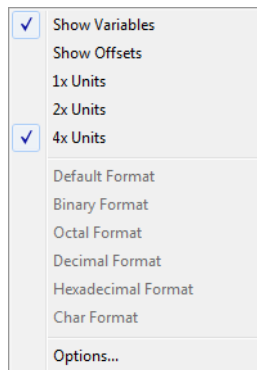
Displays the value of the variable that is displayed in the **Variable** column.

### Frame

Displays the name of the function that the call frame corresponds to.

## Context menu

This context menu is available:



These commands are available:

### Show variables

Displays separate columns named **Variables**, **Value**, and **Frame** in the **Stack** window. Variables located at memory addresses listed in the **Stack** window are displayed in these columns.

### Show offsets

Displays locations in the **Location** column as offsets from the stack pointer. When deselected, locations are displayed as absolute addresses.

### 1x Units

Displays the memory contents as single bytes.

### 2x Units

Displays the memory contents as 2-byte groups.

### 4x Units

Displays the memory contents as 4-byte groups.

**Default Format,  
Binary Format,  
Octal Format,  
Decimal Format,  
Hexadecimal Format,  
Char Format**

Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

The display format setting affects different types of expressions in these ways:

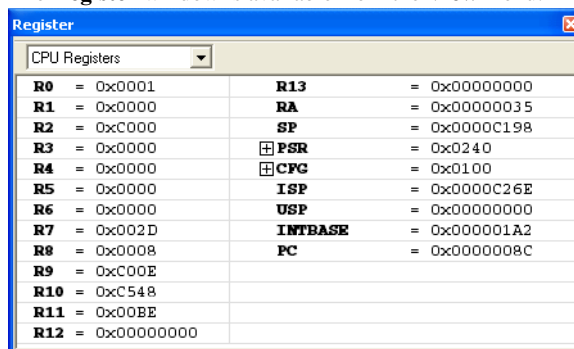
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

**Options**

Opens the **IDE Options** dialog box where you can set options specific to the **Stack** window, see the *IDE Project Management and Building Guide*.

## Register window

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



This window gives an up-to-date display of the contents of the processor registers and special function registers, and allows you to edit the content of some of the registers.

Optionally, you can choose to load either predefined register groups or to define your own application-specific groups.

You can open several instances of this window, which is very convenient if you want to keep track of different register groups.

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 59.

### To enable predefined register groups:

- 1 Select a device description file that suits your device, see *Selecting a device description file*, page 52.
- 2 The register groups appear in the **Register** window, provided that they are defined in the device description file. Note that the available register groups are also listed on the **Register Filter** page.

To define application-specific register groups:

See *Defining application-specific register groups*, page 144.

## Requirements

None; this window is always available.

## Toolbar

The toolbar contains:

### CPU Registers

Selects which register group to display, by default **CPU Registers**. Additional register groups are predefined in the device description files that make SFR registers available in the register window. The device description file contains a section that defines the special function registers and their groups.

## Display area

Displays registers and their values. Every time C-SPY stops, a value that has changed since the last stop is highlighted. Some of the registers are read-only, some of the registers are write-only (marked with *w*), and some of the registers are editable. To edit the contents of an editable register, click it, and modify its value. Press Esc to cancel the new value.

Some registers are expandable, which means that the register contains interesting bits or subgroups of bits.

To change the display format, change the **Base** setting on the **Register Filter** page—available by choosing **Tools>Options**.



For the C-SPY Simulator, these additional support registers are available in the CPU Registers group:

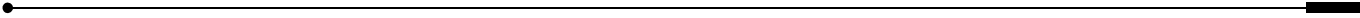
<b>CYCLES</b>	Cleared when an application is started or reset and is incremented with the number of used cycles during execution.
---------------	---



# Part 2. Analyzing your application

This part of the *C-SPY® Debugging Guide for AVR* includes these chapters:

- Trace
- Profiling
- Code coverage





# Trace

- Introduction to using trace
- Collecting and using trace data
- Reference information on trace

---

## Introduction to using trace

These topics are covered:

- Reasons for using trace
- Briefly about trace
- Requirements for using trace

See also:

- *Getting started using data logging*, page 92
- *Profiling*, page 183

### REASONS FOR USING TRACE

By using trace, you can inspect the program flow up to a specific state, for instance an application crash, and use the trace data to locate the origin of the problem. Trace data can be useful for locating programming errors that have irregular symptoms and occur sporadically.

### BRIEFLY ABOUT TRACE

To use trace in C-SPY requires that your target system can generate trace data. Once generated, C-SPY can collect it and you can visualize and analyze the data in various windows and dialog boxes.

Trace data is a continuously collected sequence of every executed instruction or data accesses for a selected portion of the execution.

### Trace features in C-SPY

In C-SPY, you can use the trace-related windows **Trace**, **Function Trace**, **Timeline**, and **Find in Trace**.

Depending on your C-SPY driver, you:

- Can set various types of trace breakpoints to control the collection of trace data.
- Have access to windows such as the **Data Log**, and **Data Log Summary**.

In addition, several other features in C-SPY also use trace data, features such as Profiling, Code coverage, and Instruction profiling.

## REQUIREMENTS FOR USING TRACE

The C-SPY simulator supports trace-related functionality, and there are no specific requirements.

Trace data cannot be collected from the hardware debugger systems.

## Collecting and using trace data

These tasks are covered:

- Getting started with trace
- Trace data collection using breakpoints
- Searching in trace data
- Browsing through trace data.

### GETTING STARTED WITH TRACE

To collect trace data using the C-SPY simulator, no specific build settings are required.

#### To get started using trace:



- 1 After you have built your application and started C-SPY, open the **Trace** window—available from the driver-specific menu—and click the **Activate** button to enable collecting trace data.
- 2 Start the execution. When the execution stops, for example because a breakpoint is triggered, trace data is displayed in the **Trace** window. For more information about the window, see *Trace window*, page 168.

### TRACE DATA COLLECTION USING BREAKPOINTS

A convenient way to collect trace data between two execution points is to start and stop the data collection using dedicated breakpoints. Choose between these alternatives:

- In the editor or **Disassembly** window, position your insertion point, right-click, and toggle a **Trace Start** or **Trace Stop** breakpoint from the context menu.
- In the **Breakpoints** window, choose **Trace Start** or **Trace Stop**.

- The C-SPY system macros `__setTraceStartBreak` and `__setTraceStopBreak` can also be used.

For more information about these breakpoints, see *Trace Start breakpoints dialog box*, page 178 and *Trace Stop breakpoints dialog box*, page 179, respectively.

## SEARCHING IN TRACE DATA

When you have collected trace data, you can perform searches in the collected data to locate the parts of your code or data that you are interested in, for example, a specific interrupt or accesses of a specific variable.

You specify the search criteria in the **Find in Trace** dialog box and view the result in the **Find in Trace** window.

The **Find in Trace** window is very similar to the **Trace** window, showing the same columns and data, but *only* those rows that match the specified search criteria.

Double-clicking an item in the **Find in Trace** window brings up the same item in the **Trace** window.

### To search in your trace data:



- 1 On the **Trace** window toolbar, click the **Find** button.
- 2 In the **Find in Trace** dialog box, specify your search criteria.

Typically, you can choose to search for:

- A specific piece of text, for which you can apply further search criteria
- An address range
- A combination of these, like a specific piece of text within a specific address range.

For more information about the various options, see *Find in Trace dialog box*, page 181.

- 3 When you have specified your search criteria, click **Find**. The **Find in Trace** window is displayed, which means you can start analyzing the trace data. For more information, see *Find in Trace window*, page 182.

## BROWSING THROUGH TRACE DATA

To follow the execution history, simply look and scroll in the **Trace** window. Alternatively, you can enter *browse mode*.



- To enter browse mode, double-click an item in the **Trace** window, or click the **Browse** toolbar button.

The selected item turns yellow and the source and disassembly windows will highlight the corresponding location. You can now move around in the trace data using the up and down arrow keys, or by scrolling and clicking; the source and **Disassembly** windows

will be updated to show the corresponding location. This is like stepping backward and forward through the execution history.

Double-click again to leave browse mode.

---

## Reference information on trace

Reference information about:

- *Trace window*, page 168
- *Function Trace window*, page 170
- *Timeline window*, page 171
- *Viewing Range dialog box*, page 177
- *Trace Start breakpoints dialog box*, page 178
- *Trace Stop breakpoints dialog box*, page 179
- *Trace Expressions window*, page 180
- *Find in Trace dialog box*, page 181
- *Find in Trace window*, page 182.

## Trace window

The **Trace** window is available from the C-SPY driver menu.

This window displays the collected trace data.

### Requirements

The C-SPY simulator.

### Trace toolbar

The toolbar in the **Trace** window and in the **Function Trace** window contains:



#### Enable/Disable

Enables and disables collecting and viewing trace data in this window. This button is not available in the **Function Trace** window.



#### Clear trace data

Clears the trace buffer. Both the **Trace** window and the **Function Trace** window are cleared.



**Toggle source**

Toggles the **Trace** column between showing only disassembly or disassembly together with the corresponding source code.

**Browse**

Toggles browse mode on or off for a selected item in the **Trace** window, see *Browsing through trace data*, page 167.

**Find**

Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 181.

**Save**

Displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.

**Edit Settings**

In the C-SPY simulator, this button is not enabled.

**Edit Expressions (C-SPY simulator only)**

Opens the **Trace Expressions** window, see *Trace Expressions window*, page 180.

**Display area**

This area displays a collected sequence of executed machine instructions. In addition, the window can display trace data for expressions.

#	Cycles	Trace	callCount
5064	13582	00044F JC 0x043C DoForegroundProcess();	5
5065	13588	00043C LCALL DoForegrou... DoForegroundProcess::?relay:	5
5066	13594	000093 18 ; '.' ?BDISPATCH_FF:	5
5067	13597	000075 POP DPH	5
5068	13600	000077 POP DPL	5
5069	13604	000079 PUSH ?CBANK	5

This area contains these columns for the C-SPY simulator:

**#**

A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.

### Cycles

The number of cycles elapsed to this point.

### Trace

The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.

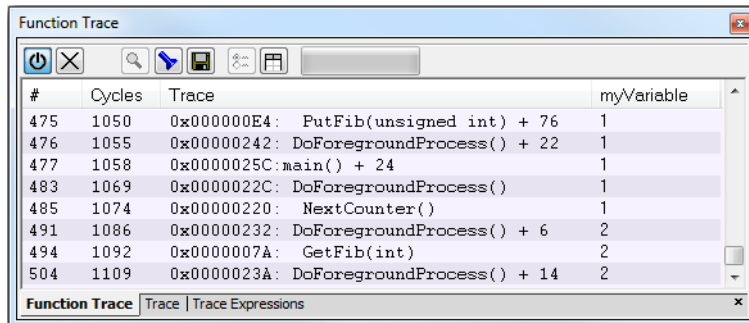
### Expression

Each expression you have defined to be displayed appears in a separate column. Each entry in the expression column displays the value *after* executing the instruction on the same row. You specify the expressions for which you want to collect trace data in the **Trace Expressions** window, see *Trace Expressions window*, page 180.

A red-colored row indicates that the previous row and the red row are not consecutive. This means that there is a gap in the collected trace data, for example because trace data has been lost due to an overflow.

## Function Trace window

The **Function Trace** window is available from the C-SPY driver menu during a debug session.



The screenshot shows a window titled "Function Trace" with a toolbar and a table of trace data. The table has four columns: "#", "Cycles", "Trace", and "myVariable". The data rows are as follows:

#	Cycles	Trace	myVariable
475	1050	0x000000E4: PutFib(unsigned int) + 76	1
476	1055	0x00000242: DoForegroundProcess() + 22	1
477	1058	0x0000025C: main() + 24	1
483	1069	0x0000022C: DoForegroundProcess()	1
485	1074	0x00000220: NextCounter()	1
491	1086	0x00000232: DoForegroundProcess() + 6	2
494	1092	0x0000007A: GetFib(int)	2
504	1109	0x0000023A: DoForegroundProcess() + 14	2

The window title bar includes "Function Trace" and standard window controls. The toolbar contains icons for power, close, search, and save. The table is scrollable, and the status bar at the bottom shows "Function Trace | Trace | Trace Expressions".

This window displays a subset of the trace data displayed in the **Trace** window. Instead of displaying all rows, the **Function Trace** window only shows trace data corresponding to calls to and returns from functions.

### Requirements

The C-SPY simulator.

### Toolbar

For information about the toolbar, see *Trace window*, page 168.

## Display area

For information about the columns in the display area, see *Trace window*, page 168

## Timeline window

The Timeline window is available from the C-SPY driver menu during a debug session.

This window displays trace data in different graphs in relation to a common time axis:

- Call Stack graph
- Data Log graph

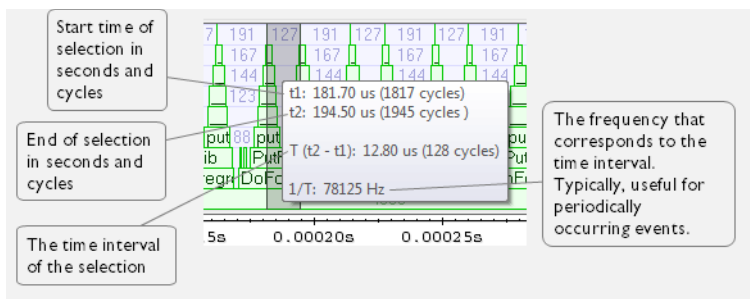
### To display a graph:

- 1 Choose **Timeline** from the C-SPY driver menu to open the Timeline window.
- 2 In the Timeline window, click in the graph area and choose **Enable** from the context menu to enable a specific graph.
- 3 Click **Go** on the toolbar to start executing your application. The graph appears.

To navigate in the graph, use any of these alternatives:

- Right-click and from the context menu choose **Zoom In** or **Zoom Out**. Alternatively, use the + and - keys. The graph zooms in or out depending on which command you used.
- Right-click in the graph and from the context menu choose **Navigate** and the appropriate command to move backwards and forwards on the graph. Alternatively, use any of the shortcut keys: arrow keys, Home, End, and Ctrl+End.
- Double-click on a sample of interest and the corresponding source code is highlighted in the editor window and in the Disassembly window.
- Click on the graph and drag to select a time interval. Press Enter or right-click and from the context menu choose **Zoom>Zoom to Selection**. The selection zooms in.

Point in the selection with the mouse pointer to get detailed tooltip information about the selected part of the graph:



Point in the graph with the mouse pointer to get detailed tooltip information for that location.

## Requirements

The display area can be populated with different graphs:

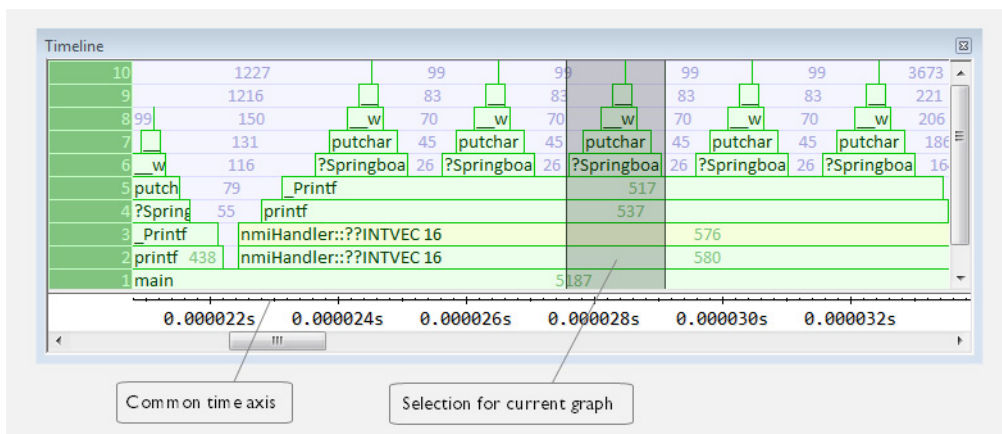
Target system	Call Stack Graph	Data Log Graph
C-SPY simulator	X	X
C-SPY hardware debugger drivers	--	--

Table 8: Supported graphs in the Timeline window

For more information about requirements related to trace data, see *Requirements for using trace*, page 166.

## Display area for the Call Stack Graph

The Call Stack Graph displays the sequence of calls and returns collected by trace.



At the bottom of the graph you will usually find `main`, and above it, the functions called from `main`, and so on. The horizontal bars, which represent invocations of functions, use four different colors:

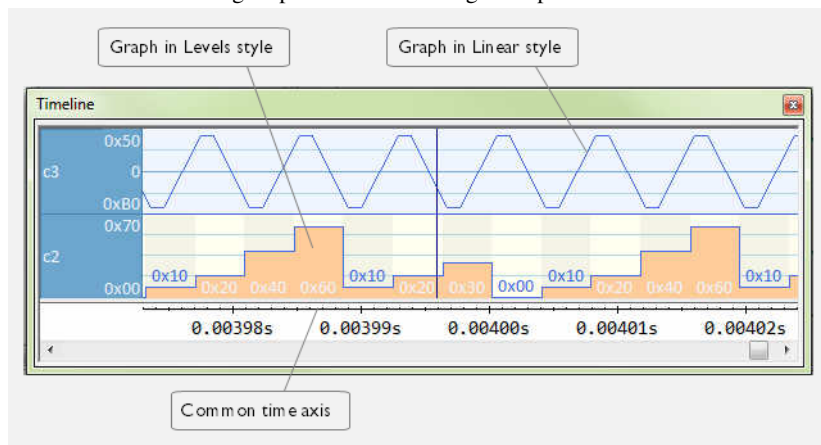
- Medium green for normal C functions with debug information
- Light green for functions known to the debugger only through an assembler label
- Medium or light yellow for interrupt handlers, with the same distinctions as for green.

The numbers represent the number of cycles spent in, or between, the function invocations.

At the bottom of the window, there is a common time axis that uses seconds as the time unit.

### Display area for the Data Log graph

The Data Log graph displays the data logs generated by trace, for up to four different variables or address ranges specified as Data Log breakpoints.



Where:

- The label area at the left end of the graph displays the variable name or the address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the Y-axis for a variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the Data Log window, see *Data Log window*, page 107.
- The graph can be displayed either as a thin line between consecutive logs or as a rectangle for every log (optionally color-filled).
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system. A red question mark indicates a log without a value.

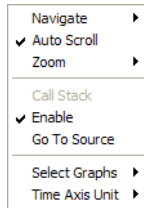
At the bottom of the window, there is a common time axis that uses seconds as the time unit.

### Selection and navigation

Click and drag to select. The selection extends vertically over all graphs, but appears highlighted in a darker color for the selected graph. You can navigate backward and forward in the selected graph using the left and right arrow keys. Use the Home and End keys to move to the first or last relevant point, respectively. Use the navigation keys in combination with the Shift key to extend the selection.

## Context menu

This context menu is available:



**Note:** The context menu contains some commands that are common to all graphs and some commands that are specific to each graph. The figure reflects the context menu for the Call Stack Graph, which means that the menu looks slightly different for the other graphs.

These commands are available:

### Navigate (All graphs)

Commands for navigating over the graph(s); choose between:

**Next** moves the selection to the next relevant point in the graph. Shortcut key: right arrow.

**Previous** moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

**First** moves the selection to the first data entry in the graph. Shortcut key: Home.

**Last** moves the selection to the last data entry in the graph. Shortcut key: End.

**End** moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

### Auto Scroll (All graphs)

Toggles auto scrolling on or off. When on, the most recently collected data is automatically displayed if you have executed the command **Navigate>End**.

### Zoom (All graphs)

Commands for zooming the window, in other words, changing the time scale; choose between:

**Zoom to Selection** makes the current selection fit the window. Shortcut key: Return.

**Zoom In** zooms in on the time scale. Shortcut key: +.

**Zoom Out** zooms out on the time scale. Shortcut key: -.

**10ns, 100ns, 1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

**1ms, 10ms**, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window.

**10m, 1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

### **Call Stack (Call Stack Graph)**

A heading that shows that the Call stack-specific commands below are available.

#### **Enable (All graphs)**

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as `OFF` in the Timeline window. If no trace data has been collected for a graph, *no data* will appear instead of the graph.

#### **Go To Source (Common)**

Displays the corresponding source code in an editor window, if applicable.

#### **Select Graphs (Common)**

Selects which graphs to be displayed in the Timeline window.

#### **Time Axis Unit (Common)**

Selects the unit used in the time axis; choose between **Seconds** and **Cycles**.

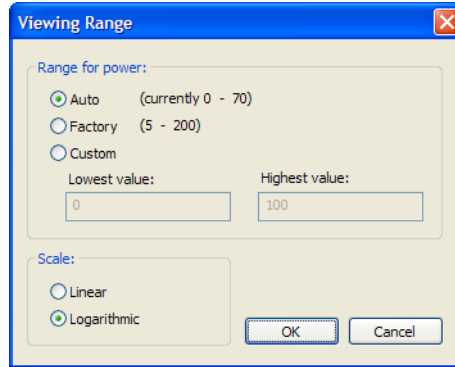
#### **Profile Selection**

Enables profiling time intervals in the Function Profiler window. Note that this command is only available if the C-SPY driver supports PC Sampling.



## Viewing Range dialog box

The **Viewing Range** dialog box is available from the context menu that appears when you right-click in the Data Log graph in the **Timeline** window.



Use this dialog box to specify the value range, that is, the range for the Y-axis for the graph.

### Requirements

The C-SPY simulator.

### Range for ...

Selects the viewing range for the displayed values:

#### Auto

Uses the range according to the range of the values that are actually collected, continuously keeping track of minimum or maximum values. The currently computed range, if any, is displayed in parentheses. The range is rounded to reasonably *even* limits.

#### Factory

For the Data Log graph: Uses the range according to the value range of the variable, for example 0–65535 for an unsigned 16-bit integer.

#### Custom

Use the text boxes to specify an explicit range.

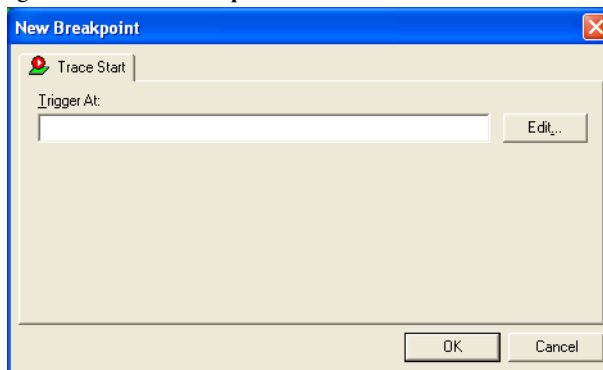
### Scale

Selects the scale type of the Y-axis:

- **Linear**
- **Logarithmic.**

## Trace Start breakpoints dialog box

The **Trace Start** dialog box is available from the context menu that appears when you right-click in the **Breakpoints** window.



Use this dialog box to set a Trace Start breakpoint where you want to start collecting trace data. If you want to collect trace data only for a specific range, you must also set a Trace Stop breakpoint where you want to stop collecting data.

See also, *Trace Stop breakpoints dialog box*, page 179.

### To set a Trace Start breakpoint:

- 1 In the editor or **Disassembly** window, right-click and choose **Trace Start** from the context menu.  
Alternatively, open the **Breakpoints** window by choosing **View>Breakpoints**.
- 2 In the **Breakpoints** window, right-click and choose **New Breakpoint>Trace Start**.  
Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.
- 3 In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 4 When the breakpoint is triggered, the trace data collection starts.

### Requirements

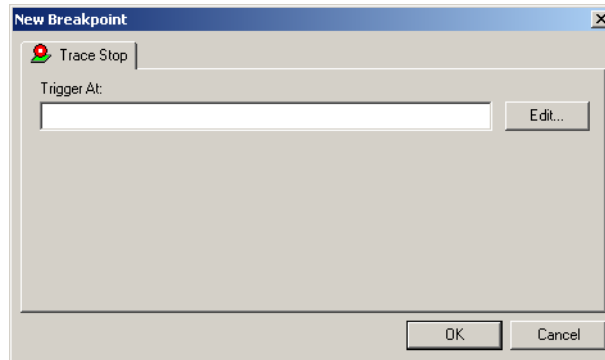
The C-SPY simulator.

### Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 138.

## Trace Stop breakpoints dialog box

The **Trace Stop** dialog box is available from the context menu that appears when you right-click in the **Breakpoints** window.



Use this dialog box to set a Trace Stop breakpoint where you want to stop collecting trace data. If you want to collect trace data only for a specific range, you might also need to set a Trace Start breakpoint where you want to start collecting data.

See also, *Trace Start breakpoints dialog box*, page 178.

### To set a Trace Stop breakpoint:

- 1 In the editor or **Disassembly** window, right-click and choose **Trace Stop** from the context menu.  
Alternatively, open the **Breakpoints** window by choosing **View>Breakpoints**.
- 2 In the **Breakpoints** window, right-click and choose **New Breakpoint>Trace Stop**.  
Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.
- 3 In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 4 When the breakpoint is triggered, the trace data collection stops.

### Requirements

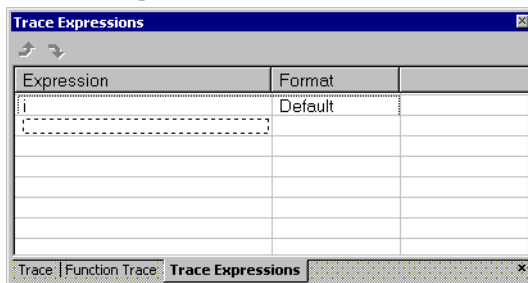
The C-SPY simulator.

### Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 138.

## Trace Expressions window

The **Trace Expressions** window is available from the **Trace** window toolbar.



Use this window to specify, for example, a specific variable (or an expression) for which you want to collect trace data.

### Requirements

The C-SPY simulator.

### Toolbar

The toolbar buttons change the order between the expressions:

#### Arrow up

Moves the selected row up.

#### Arrow down

Moves the selected row down.

### Display area

Use the display area to specify expressions for which you want to collect trace data:

#### Expression

Specify any expression that you want to collect data from. You can specify any expression that can be evaluated, such as variables and registers.

#### Format

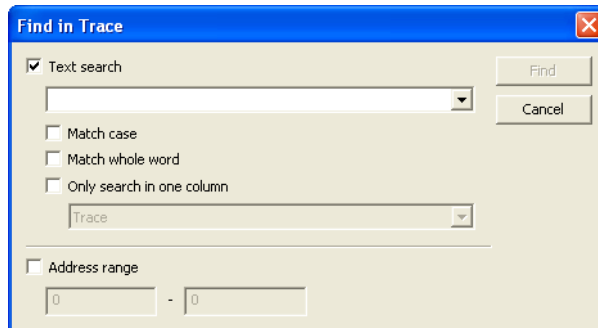
Shows which display format that is used for each expression. Note that you can change display format via the context menu.

Each row in this area will appear as an extra column in the Trace window.

## Find in Trace dialog box

The **Find in Trace** dialog box is available by clicking the **Find** button on the **Trace** window toolbar or by choosing **Edit>Find and Replace>Find**.

Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the **Trace** window is the current window or the **Find** dialog box if the editor window is the current window.



Use this dialog box to specify the search criteria for advanced searches in the trace data.

The search results are displayed in the **Find in Trace** window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 182.

See also *Searching in trace data*, page 167.

### Requirements

The C-SPY simulator.

### Text search

Specify the string you want to search for. To specify the search criteria, choose between:

#### Match Case

Searches only for occurrences that exactly match the case of the specified text. Otherwise **int** will also find **INT** and **Int** and so on.

#### Match whole word

Searches only for the string when it occurs as a separate word. Otherwise **int** will also find **print**, **sprintf** and so on.

#### Only search in one column

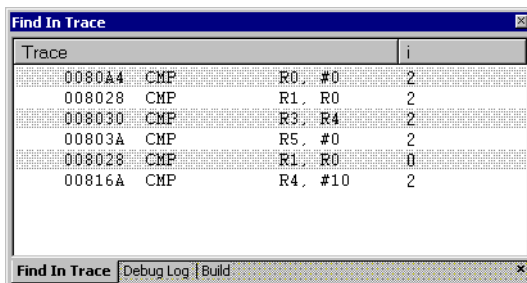
Searches only in the column you selected from the drop-down list.

## Address Range

Specify the address range you want to display or search. The trace data within the address range is displayed. If you also have specified a text string in the **Text search** field, the text string is searched for within the address range.

## Find in Trace window

The **Find in Trace** window is available from the **View>Messages** menu. Alternatively, it is automatically displayed when you perform a search using the **Find in Trace** dialog box or perform a search using the **Find in Trace** command available from the context menu in the editor window.



The screenshot shows a window titled "Find In Trace" with a table of trace data. The table has two columns: "Trace" and "i". The data rows are as follows:

Trace	i
0080A4 CMP R0, #0	2
008028 CMP R1, R0	2
008030 CMP R3, R4	2
00803A CMP R5, #0	2
008028 CMP R1, R0	0
00816A CMP R4, #10	2

The window title bar includes "Find In Trace", "Debug Log", and "Build".

This window displays the result of searches in the trace data. Double-click an item in the **Find in Trace** window to bring up the same item in the **Trace** window.

Before you can view any trace data, you must specify the search criteria in the **Find in Trace** dialog box, see *Find in Trace dialog box*, page 181.

For more information, see *Searching in trace data*, page 167.

## Requirements

The C-SPY simulator.

## Display area

The **Find in Trace** window looks like the **Trace** window and shows the same columns and data, but *only* those rows that match the specified search criteria.

# Profiling

- Introduction to the profiler
- Using the profiler
- Reference information on the profiler

---

## Introduction to the profiler

These topics are covered:

- Reasons for using the profiler
- Briefly about the profiler
- Requirements for using the profiler

### REASONS FOR USING THE PROFILER

*Function profiling* can help you find the functions in your source code where the most time is spent during execution. You should focus on those functions when optimizing your code. A simple method of optimizing a function is to compile it using speed optimization. Alternatively, you can move the data used by the function into more efficient memory. For detailed information about efficient memory usage, see the *IAR C/C++ Compiler Reference Guide for AVR*.

Alternatively, you can use *filtered profiling*, which means that you can exclude, for example, individual functions from being profiled. To profile only a specific part of your code, you can select a *time interval*—using the **Timeline** window—for which C-SPY produces profiling information.

*Instruction profiling* can help you fine-tune your code on a very detailed level, especially for assembler source code. Instruction profiling can also help you to understand where your compiled C/C++ source code spends most of its time, and perhaps give insight into how to rewrite it for better performance.

### BRIEFLY ABOUT THE PROFILER

*Function profiling* information is displayed in the **Function Profiler** window, that is, timing information for the functions in an application. Profiling must be turned on explicitly using a button on the window's toolbar, and will stay enabled until it is turned off.

*Instruction profiling* information is displayed in the **Disassembly** window, that is, the number of times each instruction has been executed.

### Profiling sources

The profiler can use different mechanisms, or *sources*, to collect profiling information. Depending on the available trace source features, one or more of the sources can be used for profiling:

- Trace (calls)
 

The full instruction trace is analyzed to determine all function calls and returns. When the collected instruction sequence is incomplete or discontinuous, the profiling information is less accurate.
- Trace (flat)
 

Each instruction in the full instruction trace or each PC Sample is assigned to a corresponding function or code fragment, without regard to function calls or returns. This is most useful when the application does not exhibit normal call/return sequences, such as when you are using an RTOS, or when you are profiling code which does not have full debug information.

### REQUIREMENTS FOR USING THE PROFILER

The C-SPY simulator support the profiler; there are no specific requirements.

---

## Using the profiler

These tasks are covered:

- Getting started using the profiler on function level
- Analyzing the profiling data
- Getting started using the profiler on instruction level

### GETTING STARTED USING THE PROFILER ON FUNCTION LEVEL



**To display function profiling information in the Function Profiler window:**

- I Build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Format>Debug information for C-SPY

*Table 9: Project options for enabling the profiler*



- 2  When you have built your application and started C-SPY, choose **C-SPY driver>Function Profiler** to open the **Function Profiler** window, and click the **Enable** button to turn on the profiler. Alternatively, choose **Enable** from the context menu that is available when you right-click in the **Function Profiler** window.
- 3 Start executing your application to collect the profiling information.
- 4 Profiling information is displayed in the **Function Profiler** window. To sort, click on the relevant column header.
- 5  When you start a new sampling, you can click the **Clear** button—alternatively, use the context menu—to clear the data.

## ANALYZING THE PROFILING DATA

Here follow some examples of how to analyze the data.

The first figure shows the result of profiling using **Source: Trace (calls)**. The profiler follows the program flow and detects function entries and exits.

- For the **InitFib** function, **Flat Time** 231 is the time spent inside the function itself.
- For the **InitFib** function, **Acc Time** 487 is the time spent inside the function itself, including all functions **InitFib** calls.
- For the **InitFib/GetFib** function, **Acc Time** 256 is the time spent inside **GetFib** (but only when called from **InitFib**), including any functions **GetFib** calls.

- Further down in the data, you can find the **GetFib** function separately and see all of its subfunctions (in this case none).

The screenshot shows the 'Function Profiler' window with a table of function calls. The table has columns for Function, Calls, Flat Time, Flat Time (%), Acc. Time, and Acc. Time (%). The 'GetFib' function is highlighted in blue, and a context menu is open over it. The menu options are: Enable (checked), Clear, Source: Trace (calls) (checked), and Source: Trace (flat). Red circles highlight the '231' value in the Flat Time column for the 'GetFib' function and the '487' value in the Acc. Time column for the 'GetFib' function.

Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
main	1	165	3.58	4356	94.39
DoForegroundProcess	10			3704	
InitFib	1			487	
PutFib	10	3174	68.78	3174	68.78
NextCounter	10	100	2.17	100	2.17
InitFib	1	231	5.01	487	10.55
GetFib	16			256	
GetFib	26	416	9.01	416	9.01
DoForegroundProcess	10	270	5.85	3704	80.26
GetFib	10			160	
NextCounter	10				
PutFib	10				
<Other>	0	25			98.85
main	1				

The second figure shows the result of profiling using **Source: Trace (flat)**. In this case, the profiler does not follow the program flow, instead the profiler only detects whether the PC address is within the function scope. For incomplete trace data, the data might contain minor errors.

For the **InitFib** function, **Flat Time** 231 is the time (number of hits) spent inside the function itself.

Function	PC Samp...	PC Samples ...
<Idle>	0	0.00
<No function>	5	0.21
DoForegroundProcess	90	3.85
GetFib	260	11.12
InitFib	141	6.03
NextCounter	60	2.57
PutFib	230	9.84
__cmain, ?main	4	0.17
__default_handler, NML_H...	0	0.00
__dwrite		
__exit		
__jar_close_ttio		
__jar_copy_init3		
__jar_data_init3		
__jar_get_ttio		
__jar_lookup_ttioh		
__jar_sh_stdout		

To secure valid data when using a debug probe, make sure to use the maximum trace buffer size and set a breakpoint in your code to stop the execution before the buffer is full.

## GETTING STARTED USING THE PROFILER ON INSTRUCTION LEVEL

**To display instruction profiling information in the Disassembly window:**

- 1 When you have built your application and started C-SPY, choose **View>Disassembly** to open the **Disassembly** window, and choose **Instruction Profiling>Enable** from the context menu that is available when you right-click in the left-hand margin of the **Disassembly** window.
- 2 Make sure that the **Show** command on the context menu is selected, to display the profiling information.
- 3 Start executing your application to collect the profiling information.
- 4 When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, you can view instruction level profiling information in the left-hand margin of the window.

The screenshot shows a window titled "Disassembly" with a dropdown menu set to "main" and "Memory". The assembly code is as follows:

```

Dly100us:
text_12:
0 08005F92 B082 SUB SP, SP, #0x8
Int32U_Dly = (Int32U)arg;
0 08005F94 E005 B ??Dly100us_0
for(volatile int i = LOOP_DLY_100US; i; i--):
??Dly100us_1:
34 08005F96 9900 LDR R1, [SP]
5 08005F98 1E49 SUBS R1, R1, #0x1
5 08005F9A 9100 STR R1, [SP]
for(volatile int i = LOOP_DLY_100US; i; i--):
??Dly100us_2:
11 08005F9C 9900 LDR R1, [SP]
3 08005F9E 2900 CMP R1, #0x0
34 08005FA0 D1F9 BNE ??Dly100us_1
while(Dly--)
??Dly100us_0:
0 08005FA2 0001 MOVS R1, R0
0 08005FA4 1E48 SUBS R0, R1, #0x1

```

The left margin of the disassembly window contains execution counts for each instruction, such as 0, 34, 5, 5, 11, 3, 34, 0, and 0.

For each instruction, the number of times it has been executed is displayed.

## Reference information on the profiler

Reference information about:

- *Function Profiler window*, page 188

See also:

- *Disassembly window*, page 76

## Function Profiler window

The **Function Profiler** window is available from the C-SPY driver menu.

The screenshot shows a window titled "Function Profiler" with a toolbar and a table of function profiling information.

Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
main()	1	165	3.57	4356	94.18
PutFib(unsigned int)	10	3174	68.63	3174	68.63
NextCounter()	10	100	2.16	100	2.16
InitFib()	1	231	4.99	487	10.53
GetFib(int)	26	416	8.99	416	8.99
DoForegroundProcess()	10	270	5.84	3704	80.09
<Other>	0	269	5.82	4572	98.85

This window displays function profiling information.

When Trace(flat) is selected, a checkbox appears on each line in the left-side margin of the window. Use these checkboxes to include or exclude lines from the profiling. Excluded lines are dimmed but not removed.

## Requirements

The C-SPY simulator.

## Toolbar

The toolbar contains:



### Enable/Disable

Enables or disables the profiler.



### Clear

Clears all profiling data.



### Save

Opens a standard **Save As** dialog box where you can save the contents of the window to a file, with tab-separated columns. Only non-expanded rows are included in the list file.



### Graphical view

Overlays the values in the percentage columns with a graphical bar.

### *Progress bar*

Displays a backlog of profiling data that is still being processed. If the rate of incoming data is higher than the rate of the profiler processing the data, a backlog is accumulated. The progress bar indicates that the profiler is still processing data, but also approximately how far the profiler has come in the process. Note that because the profiler consumes data at a certain rate and the target system supplies data at another rate, the amount of data remaining to be processed can both increase and decrease. The progress bar can grow and shrink accordingly.

## Display area

The content in the display area depends on which source that is used for the profiling information:

- For the Trace (calls) source, the display area contains one line for each function compiled with debug information enabled. When some profiling information has been collected, it is possible to expand rows of functions that have called other functions. The child items for a given function list all the functions that have been called by the parent function and the corresponding statistics.

- For the Trace (flat) source, the display area contains one line for each C function of your application, but also lines for sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels. Each executed PC address from trace data is treated as a separate sample and is associated with the corresponding line in the Profiling window. Each line contains a count of those samples.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 184.

More specifically, the display area provides information in these columns:

**Function (All sources)**

The name of the profiled C function.

**Calls (Trace (calls))**

The number of times the function has been called.

**Flat time (Trace (calls))**

The time expressed as the estimated number of cycles spent inside the function.

**Flat time (%) (Trace (calls))**

Flat time expressed as a percentage of the total time.

**Acc. time (Trace (calls))**

The time expressed as the estimated number of cycles spent inside the function and everything called by the function.

**Acc. time (%) (Trace (calls))**

Accumulated time expressed as a percentage of the total time.

**PC Samples (Trace (flat))**

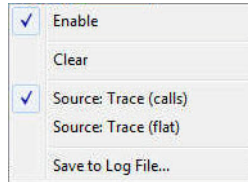
The number of PC samples associated with the function.

**PC Samples (%) (Trace (flat))**

The number of PC samples associated with the function as a percentage of the total number of samples.

## Context menu

This context menu is available:



The contents of this menu depend on the C-SPY driver you are using.

These commands are available:

### Enable

Enables the profiler. The system will collect information also when the window is closed.

### Clear

Clears all profiling data.

### Filtering

Selects which part of your code to profile. Choose between:

**Check All**—Excludes all lines from the profiling.

**Uncheck All**—Includes all lines in the profiling.

**Load**—Reads all excluded lines from a saved file.

**Save**—Saves all excluded lines to a file. Typically, this can be useful if you are a group of engineers and want to share sets of exclusions.

These commands are only available when using Trace(flat).

### Source\*

Selects which source to be used for the profiling information. Choose between:

**Trace (calls)**—the instruction count for instruction profiling is only as complete as the collected trace data.

**Trace (flat)**—the instruction count for instruction profiling is only as complete as the collected trace data.

### Save to Log File

Saves all profiling data to a file.

\* The available sources depend on the C-SPY driver you are using.





# Code coverage

- Introduction to code coverage
- Reference information on code coverage.

---

## Introduction to code coverage

These topics are covered:

- Reasons for using code coverage
- Briefly about code coverage
- Requirements and restrictions for using code coverage.

### REASONS FOR USING CODE COVERAGE

The code coverage functionality is useful when you design your test procedure to verify whether all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

### BRIEFLY ABOUT CODE COVERAGE

The **Code Coverage** window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

### REQUIREMENTS AND RESTRICTIONS FOR USING CODE COVERAGE

Code coverage is supported by the C-SPY Simulator and there are no specific requirements or restrictions.

---

## Reference information on code coverage

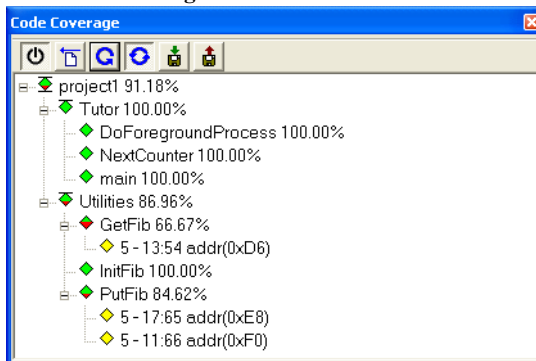
Reference information about:

- *Code Coverage window*, page 194.

See also *Single stepping*, page 70.

## Code Coverage window

The **Code Coverage** window is available from the **View** menu.



This window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

An asterisk (\*) in the title bar indicates that C-SPY has continued to execute, and that the **Code Coverage** window must be refreshed because the displayed information is no longer up to date. To update the information, use the **Refresh** command.

### To get started using code coverage:

- 1 Before using the code coverage functionality you must build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Format>Debug information for C-SPY
Debugger	Plugins>Code Coverage

Table 10: Project options for enabling code coverage

- 2 After you have built your application and started C-SPY, choose **View>Code Coverage** to open the **Code Coverage** window.
- 3 Click the **Activate** button, alternatively choose **Activate** from the context menu, to switch on code coverage.
- 4 Start the execution. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, click the **Refresh** button to view the code coverage information.



## Requirements

The C-SPY simulator.

## Display area

The code coverage information is displayed in a tree structure, showing the program, module, function, and statement levels. The window displays only source code that was compiled with debug information. Thus, startup code, exit code, and library code is not displayed in the window. Furthermore, coverage information for statements in inlined functions is not displayed. Only the statement containing the inlined function call is marked as executed. The plus sign and minus sign icons allow you to expand and collapse the structure.

These icons give you an overview of the current status on all levels:

Red diamond	Signifies that 0% of the modules or functions has been executed.
Green diamond	Signifies that 100% of the modules or functions has been executed.
Red and green diamond	Signifies that some of the modules or functions have been executed.
Yellow diamond	Signifies a statement that has not been executed.

The percentage displayed at the end of every program, module, and function line shows the amount of statements that has been covered so far, that is, the number of executed statements divided with the total number of statements.

For statements that have not been executed (yellow diamond), the information displayed is the column number range and the row number of the statement in the source window, followed by the address of the step point:

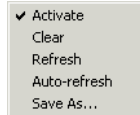
```
<column_start>-<column_end>:row address.
```

A statement is considered to be executed when one of its instructions has been executed. When a statement has been executed, it is removed from the window and the percentage is increased correspondingly.

Double-clicking a statement or a function in the **Code Coverage** window displays that statement or function as the current position in the editor window, which becomes the active window. Double-clicking a module on the program level expands or collapses the tree structure.

## Context menu

This context menu is available:



These commands are available:



### Activate

Switches code coverage on and off during execution.



### Clear

Clears the code coverage information. All step points are marked as not executed.



### Refresh

Updates the code coverage information and refreshes the window. All step points that have been executed since the last refresh are removed from the tree.



### Auto-refresh

Toggles the automatic reload of code coverage information on and off. When turned on, the code coverage information is reloaded automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.

### Save As

Saves the current code coverage result in a text file.



### Save session

Saves your code coverage session data to a \*.dat file. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command might not be supported by the C-SPY driver you are using.



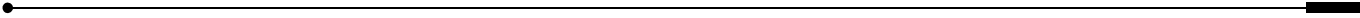
### Restore session

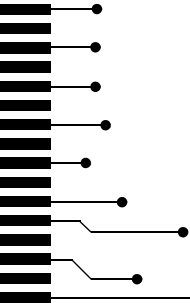
Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command might not be supported by the C-SPY driver you are using.

# Part 3. Advanced debugging

This part of the *C-SPY® Debugging Guide for AVR* includes these chapters:

- Interrupts
- C-SPY macros
- The C-SPY command line utility—`cspybat`





# Interrupts

- Introduction to interrupts
- Using the interrupt system
- Reference information on interrupts

---

## Introduction to interrupts

These topics are covered:

- Briefly about the interrupt simulation system
- Interrupt characteristics
- C-SPY system macros for interrupt simulation
- Target-adapting the interrupt simulation system

See also:

- *Reference information on C-SPY system macros*, page 221
- *Breakpoints*, page 113
- *The IAR C/C++ Compiler Reference Guide for AVR*

### **BRIEFLY ABOUT THE INTERRUPT SIMULATION SYSTEM**

By simulating interrupts, you can test the logic of your interrupt service routines and debug the interrupt handling in the target system long before any hardware is available. If you use simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices.

The C-SPY Simulator includes an interrupt simulation system where you can simulate the execution of interrupts during debugging. You can configure the interrupt simulation system so that it resembles your hardware interrupt system.

The interrupt system has the following features:

- Simulated interrupt support for the AVR microcontroller
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for various devices
- Configuration of hold time, probability, and timing variation
- State information for locating timing problems

- Configuration of interrupts using a dialog box or a C-SPY system macro—that is, one interactive and one automating interface.

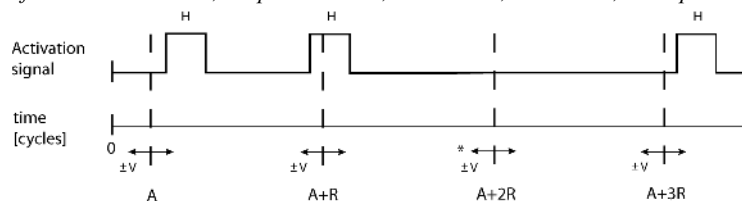
All interrupts you define using the **Interrupt Setup** dialog box exist only until they have been serviced and are not preserved between sessions.



The interrupt simulation system is activated by default, but if not required, you can turn off the interrupt simulation system to speed up the simulation. To turn it off, use either the **Interrupt Setup** dialog box or a system macro.

## INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability*.



\* If probability is less than 100%, some interrupts may be omitted.

A = Activation time  
R = Repeat interval  
H = Hold time  
V = Variance

The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation time. However, interrupts can only be raised between instructions, which means that a full assembler instruction must have been executed before the interrupt is generated, regardless of how many cycles an instruction takes.

To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—the probability, in percent, that the interrupt will actually appear in a period—and *variance*—a time variation range as a percentage of the repeat interval. These options make it possible to randomize the interrupt simulation. You can also specify a *hold time* which describes how long the interrupt remains pending until removed if it has not been processed.



## C-SPY SYSTEM MACROS FOR INTERRUPT SIMULATION

Macros are useful when you already have sorted out the details of the simulated interrupt so that it fully meets your requirements. If you write a macro function containing definitions for the simulated interrupts, you can execute the functions automatically when C-SPY starts. Another advantage is that your simulated interrupt definitions will be documented if you use macro files, and if you are several engineers involved in the development project you can share the macro files within the group.

The C-SPY Simulator provides these predefined system macros related to interrupts:

```
__enableInterrupts
__disableInterrupts
__orderInterrupt
__cancelInterrupt
__cancelAllInterrupts
```

The parameters of the first five macros correspond to the equivalent entries of the **Interrupts** dialog box.

For more information about each macro, see *Reference information on C-SPY system macros*, page 221.

## TARGET-ADAPTING THE INTERRUPT SIMULATION SYSTEM

The interrupt simulation system is easy to use. However, to take full advantage of the interrupt simulation system you should be familiar with how to adapt it for the processor you are using.

The interrupt simulation has a simplified behavior compared to the hardware. This means that the execution of an interrupt is only dependent on the status of the global interrupt enable bit.

To simulate device-specific interrupts, the interrupt system must have detailed information about each available interrupt. Except for default settings, this information is provided in the device description files. The default settings are used if no device description file has been specified.

For information about device description files, see *Selecting a device description file*, page 52.

## Using the interrupt system

These tasks are covered:

- Simulating a simple interrupt

See also:

- *Using C-SPY macros*, page 209 for details about how to use a setup file to define simulated interrupts at C-SPY startup
- The tutorial *Simulating an interrupt* in the Information Center.

### SIMULATING A SIMPLE INTERRUPT

This example demonstrates the method for simulating a timer interrupt. However, the procedure can also be used for other types of interrupts.

#### To simulate and debug an interrupt:

- 1 Assume this simple application which contains an interrupt service routine for a timer, which increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```
#include <sdtio.h>
#include <iom128.h>
#include <intrinsics.h>

volatile int ticks = 0;
void main (void)
{
    /* Add your timer setup code here */

    __enable_interrupt();          /* Enable interrupts */

    while (ticks < 100);          /* Endless loop */
    printf("Done\n");
}

/* Timer interrupt service routine */
#pragma vector = TIMERO_COMP_vect
__interrupt void basic_timer(void)
{
    ticks += 1;
}
```

- 2 Add the file to your project.
- 3 Choose **Project>Options>General Options>Device** and select **Atmega128**. A matching device description file will automatically be used.

- 4 Build your project and start the simulator.
- 5 Choose **Simulator>Interrupts** to open the **Interrupts** dialog box. Select the **Enable simulation** option to enable interrupt simulation. In the **Interrupt** drop-down list, select the `TIMER0_COMP` interrupt, and verify these settings:

Option	Settings
Activation	4000
Repeat interval	2000
Hold time	10
Probability (%)	100
Variance (%)	0

*Table 11: Timer interrupt settings*

Click **Install** and then click **OK**.

- 6 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:
  - Generate an interrupt when the cycle counter has passed 4000
  - Continuously repeat the interrupt after approximately 2000 cycles.

---

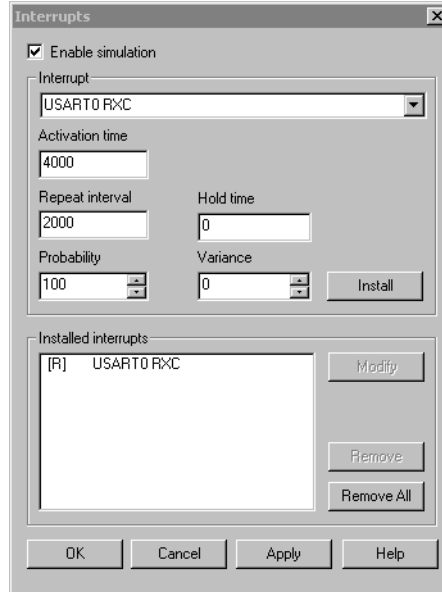
## Reference information on interrupts

Reference information about:

- *Interrupts dialog box*, page 204

## Interrupts dialog box

The **Interrupts** dialog box is available by choosing **Simulator>Interrupts**.



This dialog box lists all defined interrupts. Use this dialog box to enable or disable the interrupt simulation system, as well as to enable or disable individual interrupts.

### Requirements

The C-SPY simulator.

### Enable simulation

Enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts are generated.

### Interrupt

Lists all available interrupts. Your selection will automatically update the Description box.

The list is populated with entries from the device description file that you have selected.

### Activation time

Specify the value of the cycle counter, after which the specified type of interrupt will be generated.

**Repeat interval**

Specify the periodicity of the interrupt in cycles.

**Hold time**

Specify how long, in cycles, the interrupt remains pending until removed if it has not been processed.

**Probability**

Specify the probability, in percent, that the interrupt will actually occur within the specified period.

**Variance**

Specify a timing variation range, as a percentage of the repeat interval in which the interrupt may occur for a period. For example, if the repeat interval is 100 and the variance 5%, the interrupt might occur anywhere between  $T=95$  and  $T=105$ , to simulate a variation in the timing.

**Installed interrupts**

Lists the installed interrupts. The interrupt specification text in the list is prefixed with either [S] for a single-shot interrupt or [R] for a repeated interrupt. If the interrupt is activated but pending an additional [P] will be inserted.

**Buttons**

These buttons are available:

<b>Install</b>	Installs the interrupt you specified.
<b>Modify</b>	Edits an existing interrupt.
<b>Remove</b>	Removes the selected interrupt.
<b>Remove all</b>	Removes all installed interrupts.



# C-SPY macros

- Introduction to C-SPY macros
- Using C-SPY macros
- Reference information on the macro language
- Reference information on reserved setup macro function names
- Reference information on C-SPY system macros
- Graphical environment for macros

---

## Introduction to C-SPY macros

These topics are covered:

- Reasons for using C-SPY macros
- Briefly about using C-SPY macros
- Briefly about setup macro functions and files
- Briefly about the macro language

### REASONS FOR USING C-SPY MACROS

You can use C-SPY macros either by themselves or in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. Some examples where macros can be useful:

- Automating the debug session, for instance with trace printouts, printing values of variables, and setting breakpoints.
- Hardware configuring, such as initializing hardware registers.
- Feeding your application with simulated data during runtime.
- Simulating peripheral devices, see the chapter *Interrupts*. This only applies if you are using the simulator driver.
- Developing small debug utility functions, for instance calculating the stack depth, see the provided example `StackChk.mac` located in the directory `\AVR\src\`.

## BRIEFLY ABOUT USING C-SPY MACROS

To use C-SPY macros, you should:

- Write your macro variables and functions and collect them in one or several *macro files*
- Register your macros
- Execute your macros.

For registering and executing macros, there are several methods to choose between. Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to register or execute your macro.

## BRIEFLY ABOUT SETUP MACRO FUNCTIONS AND FILES

There are some reserved *setup macro function names* that you can use for defining macro functions which will be called at specific times, such as:

- Once after communication with the target system has been established but before downloading the application software
- Once after your application software has been downloaded
- Each time the reset command is issued
- Once when the debug session ends.

To define a macro function to be called at a specific stage, you should define and register a macro function with one of the reserved names. For instance, if you want to clear a specific memory area before you load your application software, the macro setup function `execUserPreload` should be used. This function is also suitable if you want to initialize some CPU registers or memory-mapped peripheral units before you load your application software.

You should define these functions in a *setup macro file*, which you can load before C-SPY starts. Your macro functions will then be automatically registered each time you start C-SPY. This is convenient if you want to automate the initialization of C-SPY, or if you want to register multiple setup macros.

For more information about each setup macro function, see *Reference information on reserved setup macro function names*, page 219.

## BRIEFLY ABOUT THE MACRO LANGUAGE

The syntax of the macro language is very similar to the C language. There are:

- *Macro statements*, which are similar to C statements.
- *Macro functions*, which you can define with or without parameters and return values.



- Predefined built-in *system macros*, similar to C library functions, which perform useful tasks such as opening and closing files, setting breakpoints, and defining simulated interrupts.
- *Macro variables*, which can be global or local, and can be used in C-SPY expressions.
- *Macro strings*, which you can manipulate using predefined system macros.

For more information about the macro language components, see *Reference information on the macro language*, page 214.

### Example

Consider this example of a macro function which illustrates the various components of the macro language:

```
__var oldVal;
CheckLatest(val)
{
    if (oldVal != val)
    {
        __message "Message: Changed from ", oldVal, " to ", val, "\n";
        oldVal = val;
    }
}
```

**Note:** Reserved macro words begin with double underscores to prevent name conflicts.

---

## Using C-SPY macros

These tasks are covered:

- Registering C-SPY macros—an overview
- Executing C-SPY macros—an overview
- Registering and executing using setup macros and setup files
- Executing macros using Quick Watch
- Executing a macro by connecting it to a breakpoint
- Aborting a C-SPY macro

For more examples using C-SPY macros, see:

- The tutorial about simulating an interrupt, which you can find in the Information Center
- *Initializing target hardware before C-SPY starts*, page 56.

## REGISTERING C-SPY MACROS—AN OVERVIEW

C-SPY must know that you intend to use your defined macro functions, and thus you must *register* your macros. There are various ways to register macro functions:

- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files*, page 211.
- You can register macros interactively in the **Macro Registration** window, see *Macro Registration window*, page 258. Registered macros appear in the **Debugger Macros** window, see *Debugger Macros window*, page 260.
- You can register a file containing macro function definitions, using the system macro `__registerMacroFile`. This means that you can dynamically select which macro files to register, depending on the runtime conditions. Using the system macro also lets you register multiple files at the same moment. For information about the system macro, see *\_\_registerMacroFile*, page 241.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to register your macro.

## EXECUTING C-SPY MACROS—AN OVERVIEW

There are various ways to execute macro functions:

- You can execute macro functions during the C-SPY startup sequence and at other predefined stages during the debug session by defining setup macro functions in a setup macro file, see *Registering and executing using setup macros and setup files*, page 211.
- The **Quick Watch** window lets you evaluate expressions, and can thus be used for executing macro functions. For an example, see *Executing macros using Quick Watch*, page 211.
- The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is more specified on designed for C-SPY macros. See *Macro Quicklaunch window*, page 262.
- A macro can be connected to a breakpoint; when the breakpoint is triggered the macro is executed. For an example, see *Executing a macro by connecting it to a breakpoint*, page 212.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to execute your macro.

## REGISTERING AND EXECUTING USING SETUP MACROS AND SETUP FILES

It can be convenient to register a macro file during the C-SPY startup sequence. To do this, specify a macro file which you load before starting the debug session. Your macro functions will be automatically registered each time you start the debugger.

If you use the reserved setup macro function names to define the macro functions, you can define exactly at which stage you want the macro function to be executed.

### To define a setup macro function and load it during C-SPY startup:

- 1 Create a new text file where you can define your macro function.

For example:

```
execUserSetup()
{
    ...
    __registerMacroFile("MyMacroUtils.mac");
    __registerMacroFile("MyDeviceSimulation.mac");
}
```

This macro function registers the additional macro files `MyMacroUtils.mac` and `MyDeviceSimulation.mac`. Because the macro function is defined with the function name `execUserSetup`, it will be executed directly after your application has been downloaded.

- 2 Save the file using the filename extension `mac`.
- 3 Before you start C-SPY, choose **Project>Options>Debugger>Setup**. Select **Use Setup file** and choose the macro file you just created.

The macros will now be registered during the C-SPY startup sequence.

## EXECUTING MACROS USING QUICK WATCH

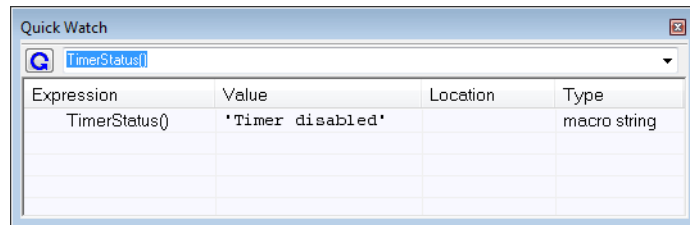
The **Quick Watch** window lets you dynamically choose when to execute a macro function.

- 1 Consider this simple macro function that checks the status of a timer enable bit:

```
TimerStatus()
{
    if ((TimerStatreg & 0x01) != 0) /* Checks the status of reg */
        return "Timer enabled"; /* C-SPY macro string used */
    else
        return "Timer disabled"; /* C-SPY macro string used */
}
```

- 2 Save the macro function using the filename extension `mac`.
- 3 To load the macro file, choose **View>Macros>Macro Registration**. The **Macro Registration** window is displayed. Click **Add** and locate the file using the file browser. The macro file appears in the list of macros in the **Macro Registration** window.
- 4 Select the macro you want to register and your macro will appear in the **Debugger Macros** window.
- 5 Choose **View>Quick Watch** to open the **Quick Watch** window, type the macro call `TimerStatus()` in the text field and press Return,

Alternatively, in the macro file editor window, select the macro function name `TimerStatus()`. Right-click, and choose **Quick Watch** from the context menu that appears.



The macro will automatically be displayed in the **Quick Watch** window.

For more information, see *Quick Watch window*, page 102.

## EXECUTING A MACRO BY CONNECTING IT TO A BREAKPOINT

You can connect a macro to a breakpoint. The macro will then be executed when the breakpoint is triggered. The advantage is that you can stop the execution at locations of particular interest and perform specific actions there.



For instance, you can easily produce log reports containing information such as how the values of variables, symbols, or registers change. To do this you might set a breakpoint on a suspicious location and connect a log macro to the breakpoint. After the execution you can study how the values of the registers have changed.

### To create a log macro and connect it to a breakpoint:

- 1 Assume this skeleton of a C function in your application source code:

```
int fact(int x)
{
    ...
}
```

- 2 Create a simple log macro function like this example:

```
logfact ()
{
  __message "fact (" ,x, " )";
}
```

The `__message` statement will log messages to the Log window.

Save the macro function in a macro file, with the filename extension `mac`.

- 3 To register the macro, choose **View>Macros>Macro Registration** to open the **Macro Registration** window and add your macro file to the list. Select the file to register it. Your macro function will appear in the **Debugger Macros** window.
- 4 To set a code breakpoint, click the **Toggle Breakpoint** button on the first statement within the function `fact` in your application source code. Choose **View>Breakpoints** to open the **Breakpoints** window. Select your breakpoint in the list of breakpoints and choose the **Edit** command from the context menu.
- 5 To connect the log macro function to the breakpoint, type the name of the macro function, `logfact ()`, in the **Action** field and click **Apply**. Close the dialog box.
- 6 Execute your application source code. When the breakpoint is triggered, the macro function will be executed. You can see the result in the **Log** window.
  - Note that the expression in the **Action** field is evaluated only when the breakpoint causes the execution to really stop. If you want to log a value and then automatically continue execution, you can either:
    - Use a **Log** breakpoint, see *Log breakpoints dialog box*, page 129
    - Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 123.
- 7 You can easily enhance the log macro function by, for instance, using the `__fmessage` statement instead, which will print the log information to a file. For information about the `__fmessage` statement, see *Formatted output*, page 217.

For an example where a serial port input buffer is simulated using the method of connecting a macro to a breakpoint, see the tutorial *Simulating an interrupt* in the Information Center.

## ABORTING A C-SPY MACRO

### To abort a C-SPY macro:

- 1 Press **Ctrl+Shift+.** (period) for a short while.
- 2 A message that says that the macro has terminated is displayed in the **Debug Log** window.

This method can be used if you suspect that something is wrong with the execution, for example because it seems not to terminate in a reasonable time.

---

## Reference information on the macro language

Reference information about:

- *Macro functions*, page 214
- *Macro variables*, page 214
- *Macro parameters*, page 215
- *Macro strings*, page 215
- *Macro statements*, page 216
- *Formatted output*, page 217.

### MACRO FUNCTIONS

C-SPY macro functions consist of C-SPY variable definitions and macro statements which are executed when the macro is called. An unlimited number of parameters can be passed to a macro function, and macro functions can return a value on exit.

A C-SPY macro has this form:

```
macroName (parameterList)
{
    macroBody
}
```

where *parameterList* is a list of macro parameters separated by commas, and *macroBody* is any series of C-SPY variable definitions and C-SPY statements.

Type checking is neither performed on the values passed to the macro functions nor on the return value.

### MACRO VARIABLES

A macro variable is a variable defined and allocated outside your application. It can then be used in a C-SPY expression, or you can assign application data—values of the variables in your application—to it. For more information about C-SPY expressions, see *C-SPY expressions*, page 88.

The syntax for defining one or more macro variables is:

```
__var nameList;
```

where *nameList* is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and it exists throughout the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and destroyed on return from the macro.

By default, macro variables are treated as signed integers and initialized to 0. When a C-SPY variable is assigned a value in an expression, it also acquires the type of that expression. For example:

Expression	What it means
<code>myvar = 3.5;</code>	<code>myvar</code> is now type <code>double</code> , value <code>3.5</code> .
<code>myvar = (int*)i;</code>	<code>myvar</code> is now type <code>pointer to int</code> , and the value is the same as <code>i</code> .

Table 12: Examples of C-SPY macro variables

In case of a name conflict between a C symbol and a C-SPY macro variable, C-SPY macro variables have a higher precedence than C variables. Note that macro variables are allocated on the debugger host and do not affect your application.

## MACRO PARAMETERS

A macro parameter is intended for parameterization of device support. The named parameter will behave as a normal C-SPY macro variable with these differences:

- The parameter definition can have an initializer
- Values of a parameters can be set through options (either in the IDE or in `cs Spybat`).
- A value set from an option will take precedence over a value set by an initializer
- A parameter must have an initializer, be set through an option, or both. Otherwise, it has an undefined value, and accessing it will cause a runtime error.

The syntax for defining one or more macro parameters is:

```
__param param[ = value, ...;]
```

Use the command line option `--macro_param` to specify a value to a parameter, see `--macro_param`, page 286.

## MACRO STRINGS

In addition to C types, macro variables can hold values of *macro strings*. Note that macro strings differ from C language strings.

When you write a string literal, such as `"Hello!"`, in a C-SPY expression, the value is a macro string. It is not a C-style character pointer `char*`, because `char*` must point to a sequence of characters in target memory and C-SPY cannot expect any string literal to actually exist in target memory.

You can manipulate a macro string using a few built-in macro functions, for example `__strFind` or `__subString`. The result can be a new macro string. You can

concatenate macro strings using the + operator, for example `str + "tail"`. You can also access individual characters using subscription, for example `str[3]`. You can get the length of a string using `sizeof(str)`. Note that a macro string is not NULL-terminated.

The macro function `__toString` is used for converting from a NULL-terminated C string in your application (`char*` or `char[]`) to a macro string. For example, assume this definition of a C string in your application:

```
char const *cstr = "Hello";
```

Then examine these macro examples:

```
__var str;          /* A macro variable */
str = cstr          /* str is now just a pointer to char */
sizeof str         /* same as sizeof (char*), typically 2 or 4 */
str = __toString(cstr,512) /* str is now a macro string */
sizeof str        /* 5, the length of the string */
str[1]            /* 101, the ASCII code for 'e' */
str += " World!" /* str is now "Hello World!" */
```

See also *Formatted output*, page 217.

## MACRO STATEMENTS

Statements are expected to behave in the same way as the corresponding C statements would do. The following C-SPY macro statements are accepted:

### Expressions

```
expression;
```

For more information about C-SPY expressions, see *C-SPY expressions*, page 88.

### Conditional statements

```
if (expression)
    statement
```

```
if (expression)
    statement
else
    statement
```



## Loop statements

```
for (init_expression; cond_expression; update_expression)
    statement
```

```
while (expression)
    statement
```

```
do
    statement
while (expression);
```

## Return statements

```
return;
```

```
return expression;
```

If the return value is not explicitly set, signed int 0 is returned by default.

## Blocks

Statements can be grouped in blocks.

```
{
    statement1
    statement2
    .
    .
    statementN
}
```

## FORMATTED OUTPUT

C-SPY provides various methods for producing formatted output:

<code>__message <i>argList</i>;</code>	Prints the output to the <b>Debug Log</b> window.
<code>__fmessage <i>file</i>, <i>argList</i>;</code>	Prints the output to the designated file.
<code>__smessage <i>argList</i>;</code>	Returns a string containing the formatted output.

where *argList* is a comma-separated list of C-SPY expressions or strings, and *file* is the result of the `__openFile` system macro, see `__openFile`, page 236.

To produce messages in the **Debug Log** window:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2,
" in the Log window.";
```

This produces this message in the Log window:

This line prints the values 42 and 37 in the Log window.

To write the output to a designated file:

```
__fmessage myfile, "Result is ", res, "!\n";
```

To produce strings:

```
myMacroVar = __smessage 42, " is the answer.";
myMacroVar now contains the string "42 is the answer.".
```

### Specifying display format of arguments

To override the default display format of a scalar argument (number or pointer) in *argList*, suffix it with a `:` followed by a format specifier. Available specifiers are:

<code>%b</code>	for binary scalar arguments
<code>%o</code>	for octal scalar arguments
<code>%d</code>	for decimal scalar arguments
<code>%x</code>	for hexadecimal scalar arguments
<code>%c</code>	for character scalar arguments

These match the formats available in the **Watch** and **Locals** windows, but number prefixes and quotes around strings and characters are not printed. Another example:

```
__message "The character '", cvar:%c, "' has the decimal value
", cvar;
```

Depending on the value of the variables, this produces this message:

The character 'A' has the decimal value 65

**Note:** A character enclosed in single quotes (a character literal) is an integer constant and is not automatically formatted as a character. For example:

```
__message 'A', " is the numeric value of the character ",
'A':%c;
```

would produce:

```
65 is the numeric value of the character A
```

**Note:** The default format for certain types is primarily designed to be useful in the **Watch** window and other related windows. For example, a value of type `char` is formatted as `'A' (0x41)`, while a pointer to a character (potentially a C string) is formatted as `0x8102 "Hello"`, where the string part shows the beginning of the string (currently up to 60 characters).

When printing a value of type `char*`, use the `%x` format specifier to print just the pointer value in hexadecimal notation, or use the system macro `__toString` to get the full string value.

---

## Reference information on reserved setup macro function names

There are reserved setup macro function names that you can use for defining your setup macro functions. By using these reserved names, your function will be executed at defined stages during execution. For more information, see *Briefly about setup macro functions and files*, page 208.

Reference information about:

- `execUserPreload`
- `execUserExecutionStarted`
- `execUserExecutionStopped`
- `execUserSetup`
- `execUserPreReset`
- `execUserReset`
- `execUserExit`

### **execUserPreload**

Syntax	<code>execUserPreload</code>
For use with	All C-SPY drivers.
Description	Called after communication with the target system is established but before downloading the target application.  Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.

## execUserExecutionStarted

Syntax	<code>execUserExecutionStarted</code>
For use with	All C-SPY drivers.
Description	Called when the debugger is about to start or resume execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the <b>Disassembly</b> window.

## execUserExecutionStopped

Syntax	<code>execUserExecutionStopped</code>
For use with	All C-SPY drivers.
Description	Called when the debugger has stopped execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the <b>Disassembly</b> window.

## execUserSetup

Syntax	<code>execUserSetup</code>
For use with	All C-SPY drivers.
Description	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.



If you define interrupts or breakpoints in a macro file that is executed at system start (using `execUserSetup`) we strongly recommend that you also make sure that they are removed at system shutdown (using `execUserExit`). An example is available in `SetupSimple.mac`, see the tutorials in the Information Center.

The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time `execUserSetup` is executed again. This seriously affects the execution speed.

## execUserPreReset

Syntax	<code>execUserPreReset</code>
For use with	All C-SPY drivers.
Description	Called each time just before the reset command is issued. Implement this macro to set up any required device state.

## execUserReset

Syntax	<code>execUserReset</code>
For use with	All C-SPY drivers.
Description	Called each time just after the reset command is issued. Implement this macro to set up and restore data.

## execUserExit

Syntax	<code>execUserExit</code>
For use with	All C-SPY drivers.
Description	Called once when the debug session ends. Implement this macro to save status data etc.

---

## Reference information on C-SPY system macros

This section gives reference information about each of the C-SPY system macros.

This table summarizes the pre-defined system macros:

Macro	Description
<code>__cancelAllInterrupts</code>	Cancels all ordered interrupts
<code>__cancelInterrupt</code>	Cancels an interrupt
<code>__clearBreak</code>	Clears a breakpoint
<code>__closeFile</code>	Closes a file that was opened by <code>__openFile</code>

*Table 13: Summary of system macros*

Macro	Description
<code>__delay</code>	Delays execution
<code>__disableInterrupts</code>	Disables generation of interrupts
<code>__driverType</code>	Verifies the driver type
<code>__enableInterrupts</code>	Enables generation of interrupts
<code>__evaluate</code>	Interprets the input string as an expression and evaluates it.
<code>__fillMemory8</code>	Fills a specified memory area with a byte value.
<code>__fillMemory16</code>	Fills a specified memory area with a 2-byte value.
<code>__fillMemory32</code>	Fills a specified memory area with a 4-byte value.
<code>__getCycleCounter</code>	Reads the cycle counter
<code>__isBatchMode</code>	Checks if C-SPY is running in batch mode or not.
<code>__loadImage</code>	Loads an image.
<code>__memoryRestore</code>	Restores the contents of a file to a specified memory zone
<code>__memorySave</code>	Saves the contents of a specified memory area to a file
<code>__memoryRestoreFromFile</code>	Reads from a file and restores to memory
<code>__memorySaveToFile</code>	Saves a range of a memory zone to a file
<code>__messageBoxYesCancel</code>	Displays a Yes/Cancel dialog box for user interaction
<code>__messageBoxYesNo</code>	Displays a Yes/No dialog box for user interaction
<code>__openFile</code>	Opens a file for I/O operations
<code>__orderInterrupt</code>	Generates an interrupt
<code>__readFile</code>	Reads from the specified file
<code>__readFileByte</code>	Reads one byte from the specified file
<code>__readMemory8,</code> <code>__readMemoryByte</code>	Reads one byte from the specified memory location
<code>__readMemory16</code>	Reads two bytes from the specified memory location
<code>__readMemory32</code>	Reads four bytes from the specified memory location
<code>__registerMacroFile</code>	Registers macros from the specified file
<code>__resetFile</code>	Rewinds a file opened by <code>__openFile</code>
<code>__setCodeBreak</code>	Sets a code breakpoint

Table 13: Summary of system macros (Continued)

Macro	Description
<code>__setComplexBreak</code>	Sets a complex breakpoint
<code>__setDataBreak</code>	Sets a data breakpoint
<code>__setLogBreak</code>	Sets a log breakpoint
<code>__setSimBreak</code>	Sets a simulation breakpoint
<code>__setTraceStartBreak</code>	Sets a trace start breakpoint
<code>__setTraceStopBreak</code>	Sets a trace stop breakpoint
<code>__sourcePosition</code>	Returns the file name and source location if the current execution location corresponds to a source location
<code>__strFind</code>	Searches a given string for the occurrence of another string
<code>__subString</code>	Extracts a substring from another string
<code>__targetDebuggerVersion</code>	Returns the version of the target debugger
<code>__toLowerCase</code>	Returns a copy of the parameter string where all the characters have been converted to lower case
<code>__toString</code>	Prints strings
<code>__toUpperCase</code>	Returns a copy of the parameter string where all the characters have been converted to upper case
<code>__transparent</code>	Sends a command to the ROM-monitor transparently from C-SPY
<code>__unloadImage</code>	Unloads a debug image
<code>__writeFile</code>	Writes to the specified file
<code>__writeFileByte</code>	Writes one byte to the specified file
<code>__writeMemory8,</code> <code>__writeMemoryByte</code>	Writes one byte to the specified memory location
<code>__writeMemory16</code>	Writes a two-byte word to the specified memory location
<code>__writeMemory32</code>	Writes a four-byte word to the specified memory location

Table 13: Summary of system macros (Continued)

## `__cancelAllInterrupts`

Syntax	<code>__cancelAllInterrupts()</code>
Return value	<code>int 0</code>

For use with           The C-SPY Simulator.  
 Description           Cancels all ordered interrupts.

## **\_\_cancelInterrupt**

Syntax                `__cancelInterrupt(interrupt_id)`  
 Parameters            `interrupt_id`  
                       The value returned by the corresponding `__orderInterrupt` macro call (unsigned long).  
 Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

*Table 14: \_\_cancelInterrupt return values*

For use with           The C-SPY Simulator.  
 Description           Cancels the specified interrupt.

## **\_\_clearBreak**

Syntax                `__clearBreak(break_id)`  
 Parameters            `break_id`  
                       The value returned by any of the set breakpoint macros.  
 Return value           int 0  
 For use with           All C-SPY drivers.  
 Description           Clears a user-defined breakpoint.  
 See also              *Breakpoints*, page 113.



## \_\_closeFile

Syntax	<code>__closeFile(<i>fileHandle</i>)</code>
Parameters	<i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Closes a file previously opened by <code>__openFile</code> .

## \_\_delay

Syntax	<code>__delay(<i>value</i>)</code>
Parameters	<i>value</i> The number of milliseconds to delay execution.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Delays execution the specified number of milliseconds.

## \_\_disableInterrupts

Syntax `__disableInterrupts()`

Return value

Result	Value
Successful	<code>int 0</code>
Unsuccessful	Non-zero error number

Table 15: `__disableInterrupts` return values

For use with The C-SPY Simulator.

Description Disables the generation of interrupts.

## \_\_driverType

Syntax

```
__driverType(driver_id)
```

Parameters

*driver\_id*

A string corresponding to the driver you want to check for. Choose one of these:

"sim" corresponds to the simulator driver.

"jtagice" corresponds to the C-SPY JTAGICE driver

"jtagicemkII" corresponds to the C-SPY JTAGICE mkII driver

"atmel-ice" corresponds to the C-SPY Atmel-ICE driver

"jtagice3" corresponds to the C-SPY JTAGICE3 driver

"avrone" corresponds to the C-SPY AVR ONE! driver

"ice200" corresponds to the C-SPY ICE200 driver

"ccr" corresponds to the C-SPY CCR driver

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 16: \_\_driverType return values

For use with

All C-SPY drivers

Description

Checks to see if the current C-SPY driver is identical to the driver type of the *driver\_id* parameter.

Example

```
__driverType("sim")
```

If the simulator is the current driver, the value 1 is returned. Otherwise 0 is returned.

## \_\_enableInterrupts

Syntax

```
__enableInterrupts()
```

Return value

Result	Value
Successful	int 0

Table 17: \_\_enableInterrupts return values

Result	Value
Unsuccessful	Non-zero error number

Table 17: `__enableInterrupts` return values

For use with	The C-SPY Simulator.
Description	Enables the generation of interrupts.

## `__evaluate`

Syntax	<code>__evaluate(string, valuePtr)</code>
Parameters	<p><i>string</i> Expression string.</p> <p><i>valuePtr</i> Pointer to a macro variable storing the result.</p>

Return value

Result	Value
Successful	int 0
Unsuccessful	int 1

Table 18: `__evaluate` return values

For use with	All C-SPY drivers.
Description	This macro interprets the input string as an expression and evaluates it. The result is stored in a variable pointed to by <i>valuePtr</i> .

Example This example assumes that the variable `i` is defined and has the value 5:

```
__evaluate("i + 3", &myVar)
```

The macro variable `myVar` is assigned the value 8.

## `__fillMemory8`

Syntax	<code>__fillMemory8(value, address, zone, length, format)</code>
Parameters	<p><i>value</i> An integer that specifies the value.</p>

*address*

An integer that specifies the memory start address.

*zone*A string that specifies the memory zone, see *C-SPY memory zones*, page 142.*length*

An integer that specifies how many bytes are affected.

*format*

One of these alternatives:

Copy      *value* will be copied to the specified memory area.AND      An AND operation will be performed between *value* and the existing contents of memory before writing the result to memory.OR      An OR operation will be performed between *value* and the existing contents of memory before writing the result to memory.XOR      An XOR operation will be performed between *value* and the existing contents of memory before writing the result to memory.

Return value	int 0
For use with	All C-SPY drivers.
Description	Fills a specified memory area with a byte value.
Example	<code>__fillMemory8(0x80, 0x700, "DATA", 0x10, "OR");</code>

## **\_\_fillMemory16**

Syntax	<code>__fillMemory16(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>
--------	--

Parameters	<i>value</i>
------------	--------------

An integer that specifies the value.

*address*

An integer that specifies the memory start address.

*zone*A string that specifies the memory zone, see *C-SPY memory zones*, page 142.

	<i>length</i>	An integer that defines how many 2-byte entities to be affected.
	<i>format</i>	One of these alternatives: <ul style="list-style-type: none"> <li>Copy      <i>value</i> will be copied to the specified memory area.</li> <li>AND        An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</li> <li>OR         An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</li> <li>XOR        An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</li> </ul>
Return value	int	0
For use with		All C-SPY drivers.
Description		Fills a specified memory area with a 2-byte value.
Example		<code>__fillMemory16(0xCDCD, 0x7000, "DATA", 0x200, "Copy");</code>

## **\_\_fillMemory32**

Syntax	<code>__fillMemory32(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>
Parameters	<p><i>value</i></p> <p>An integer that specifies the value.</p> <p><i>address</i></p> <p>An integer that specifies the memory start address.</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 142.</p> <p><i>length</i></p> <p>An integer that defines how many 4-byte entities to be affected.</p>

*format*

One of these alternatives:

Copy	<i>value</i> will be copied to the specified memory area.
AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.

Return value	int 0
For use with	All C-SPY drivers.
Description	Fills a specified memory area with a 4-byte value.
Example	<code>__fillMemory32(0x0000FFFF, 0x4000, "DATA", 0x1000, "XOR");</code>

**\_\_getCycleCounter**

Syntax	<code>__getCycleCounter()</code>
Return value	Returns the current value of the cycle counter as a long long int.
For use with	The C-SPY hardware drivers.
Description	Reads the current value of the cycle counter.

**\_\_isBatchMode**

Syntax	<code>__isBatchMode()</code>
--------	------------------------------

Return value

Result	Value
True	int 1
False	int 0

Table 19: `__isBatchMode` return values

For use with

All C-SPY drivers.

Description

This macro returns True if the debugger is running in batch mode, otherwise it returns False.

## `__loadImage`

Syntax

```
__loadImage(path, offset, debugInfoOnly)
```

Parameters

*path*

A string that identifies the path to the image to download. The path must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

*offset*

An integer that identifies the offset to the destination address for the downloaded image.

*debugInfoOnly*

A non-zero integer value if no code or data should be downloaded to the target system, which means that C-SPY will only read the debug information from the debug file. Or, 0 (zero) for download.

Return value

Value	Result
Non-zero integer number	A unique module identification.
int 0	Loading failed.

Table 20: `__loadImage` return values

For use with

All C-SPY drivers.

Description

Loads an image (debug file).

Example 1

Your system consists of a ROM library and an application. The application is your active project, but you have a debug file corresponding to the library. In this case you can add

this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage(ROMfile, 0x8000, 1);
```

This macro call loads the debug information for the ROM library `ROMfile` without downloading its contents (because it is presumably already in ROM). Then you can debug your application together with the library.

#### Example 2

Your system consists of a ROM library and an application, but your main concern is the library. The library needs to be programmed into flash memory before a debug session. While you are developing the library, the library project must be the active project in the IDE. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage(ApplicationFile, 0x8000, 0);
```

The macro call loads the debug information for the application and downloads its contents (presumably into RAM). Then you can debug your library together with the application.

See also

*Images*, page 296 and *Loading multiple images*, page 55.

## \_\_memoryRestore

Syntax

```
__memoryRestore(zone, filename)
```

Parameters

*zone*

A string that specifies the memory zone, see *C-SPY memory zones*, page 142.

*filename*

A string that specifies the file to be read. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

Return value

0 if successful, otherwise 1

For use with

All C-SPY drivers.

Description

Reads the contents of a file and saves it to the specified memory zone. It is recommended that you use this macro instead of `__memoryRestoreFromFile`.

Example

```
__memoryRestore("DATA", "c:\\temp\\saved_memory.hex");
```



See also *Memory Restore dialog box*, page 151.

## \_\_memoryRestoreFromFile

Syntax	<code>__memoryRestoreFromFile(<i>filename</i>, <i>zone</i>)</code>	
Parameters	<i>filename</i>	The file to be read.
	<i>zone</i>	The memory zone name (string); for a list of available zones, see <i>C-SPY memory zones</i> , page 142.
Return value	0 if successful, otherwise 1	
For use with	All C-SPY drivers.	
Description	Reads the contents of a file in intel-hex or Motorola S-record format and writes it to the specified memory zone. This macro is available for backwards compatibility.	
Example	<code>__memoryRestoreFromFile("C:\\temp\\tmp.hex", "DATA");</code>	

## \_\_memorySave

Syntax	<code>__memorySave(<i>start</i>, <i>stop</i>, <i>format</i>, <i>filename</i>)</code>	
Parameters	<i>start</i>	A string that specifies the first location of the memory area to be saved.
	<i>stop</i>	A string that specifies the last location of the memory area to be saved.
	<i>format</i>	A string that specifies the format to be used for the saved memory. Choose between:
	<code>intel-extended</code>	
	<code>motorola</code>	
	<code>motorola-s19</code>	
	<code>motorola-s28</code>	
	<code>motorola-s37</code>	

	<i>filename</i>
	A string that specifies the file to write to. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide</i> .
Return value	0 if successful. At failure, macro execution is aborted and log messages are produced.
For use with	All C-SPY drivers.
Description	Saves the contents of a specified memory area to a file. It is recommended that you use this macro instead of <code>__memorySaveToFile</code> .
Example	<pre>__memorySave("DATA:0x00", "DATA:0xFF", "intel-extended", "c:\\temp\\saved_memory.hex");</pre>
See also	<i>Memory Save dialog box</i> , page 150.

## **\_\_memorySaveToFile**

Syntax	<code>__memorySaveToFile(filename, zone, start, stop)</code>
Parameters	
	<i>filename</i> The file to be written.
	<i>zone</i> The memory zone name (string); for a list of available zones, see <i>C-SPY memory zones</i> , page 142.
	<i>start</i> The start address of the memory range to be saved.
	<i>stop</i> The stop address of the memory range to be saved.
Return value	0 if successful, otherwise 1
For use with	All C-SPY drivers.
Description	Saves a range of a memory zone to a file. The file is written in the Intel hex format. This macro is available for backwards compatibility.
Example	<pre>__memoryRestoreFromFile("C:\\temp\\tmp.hex", "DATA", "0x1000", "0x1100");</pre>

## \_\_messageBoxYesCancel

Syntax `__messageBoxYesCancel (string message, string caption)`

Parameters *message*

A message that will appear in the message box.

*caption*

The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 21: \_\_messageBoxYesCancel return values

For use with All C-SPY drivers.

Description

Displays a Yes/Cancel dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

## \_\_messageBoxYesNo

Syntax `__messageBoxYesNo (string message, string caption)`

Parameters *message*

A message that will appear in the message box.

*caption*

The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 22: \_\_messageBoxYesNo return values

For use with All C-SPY drivers.

Description

Displays a Yes/No dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

## \_\_openFile

### Syntax

```
__openFile(filename, access)
```

### Parameters

*filename*

The file to be opened. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide*.

*access*

The access type (string).

These are mandatory but mutually exclusive:

"a" append, new data will be appended at the end of the open file

"r" read (by default in text mode; combine with b for binary mode: rb)

"w" write (by default in text mode; combine with b for binary mode: wb)

These are optional and mutually exclusive:

"b" binary, opens the file in binary mode

"t" ASCII text, opens the file in text mode

This access type is optional:

"+" together with r, w, or a; r+ or w+ is *read* and *write*, while a+ is *read* and *append*

### Return value

Result	Value
Successful	The file handle
Unsuccessful	An invalid file handle, which tests as False

Table 23: \_\_openFile return values

### For use with

All C-SPY drivers.

### Description

Opens a file for I/O operations. The default base directory of this macro is where the currently open project file (\*.ewp) is located. The argument to \_\_openFile can specify a location relative to this directory. In addition, you can use argument variables such as \$PROJ\_DIR\$ and \$TOOLKIT\_DIR\$ in the path argument.

**Example**

```

__var myFileHandle;          /* The macro variable to contain */
                             /* the file handle */
myFileHandle = __openFile("$PROJ_DIR$\\Debug\\Exe\\test.tst",
"r");
if (myFileHandle)
{
    /* successful opening */
}

```

**See also** For information about argument variables, see the *IDE Project Management and Building Guide*.

## \_\_orderInterrupt

**Syntax** `__orderInterrupt(specification, first_activation,  
repeat_interval, variance,  
hold_time, probability)`

**Parameters**

*specification*

The interrupt name (string). The interrupt system will automatically get the description from the device description file.

*first\_activation*

The first activation time in cycles (integer)

*repeat\_interval*

The periodicity in cycles (integer)

*variance*

The timing variation range in percent (integer between 0 and 100)

*hold\_time*

The hold time (integer)

*probability*

The probability in percent (integer between 0 and 100)

**Return value** The macro returns an interrupt identifier (unsigned long).

If the syntax of *specification* is incorrect, it returns -1.

**For use with** The C-SPY Simulator.

**Description** Generates an interrupt.

**Example** This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles:

```
__orderInterrupt( "USART0 TXC", 4000, 2000, 0, 0, 100 );
```

## \_\_readFile

**Syntax**

```
__readFile(fileHandle, valuePtr)
```

**Parameters**

*fileHandle*

A macro variable used as filehandle by the `__openFile` macro.

*valuePtr*

A pointer to a variable.

**Return value**

Result	Value
Successful	0
Unsuccessful	Non-zero error number

Table 24: `__readFile` return values

**For use with**

All C-SPY drivers.

**Description**

Reads a sequence of hexadecimal digits from the given file and converts them to an unsigned long which is assigned to the *value* parameter, which should be a pointer to a macro variable.

Only printable characters representing hexadecimal digits and white-space characters are accepted, no other characters are allowed.

**Example**

```
__var number;
if (__readFile(myFileHandle, &number) == 0)
{
    // Do something with number
}
```

In this example, if the file pointed to by `myFileHandle` contains the ASCII characters `1234 abcd 90ef`, consecutive reads will assign the values `0x1234 0xabcd 0x90ef` to the variable `number`.

**\_\_readFileByte**

Syntax	<code>__readFileByte(<i>fileHandle</i>)</code>
Parameters	<i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro.
Return value	-1 upon error or end-of-file, otherwise a value between 0 and 255.
For use with	All C-SPY drivers.
Description	Reads one byte from a file.
Example	<pre>__var byte; while ( (byte = __readFileByte(myFileHandle)) != -1 ) {     /* Do something with byte */ }</pre>

**\_\_readMemory8, \_\_readMemoryByte**

Syntax	<code>__readMemory8(<i>address</i>, <i>zone</i>)</code> <code>__readMemoryByte(<i>address</i>, <i>zone</i>)</code>
Parameters	<i>address</i> The memory address (integer). <i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 142.
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads one byte from a given memory location.
Example	<code>__readMemory8(0x0108, "DATA");</code>

**\_\_readMemory16**

Syntax	<code>__readMemory16(<i>address</i>, <i>zone</i>)</code>
Parameters	<p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 142.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads a two-byte word from a given memory location.
Example	<code>__readMemory16(0x0108, "DATA");</code>

**\_\_readMemory32**

Syntax	<code>__readMemory32(<i>address</i>, <i>zone</i>)</code>
Parameters	<p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 142.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads a four-byte word from a given memory location.
Example	<code>__readMemory32(0x0108, "DATA");</code>



## \_\_registerMacroFile

Syntax	<code>__registerMacroFile(<i>filename</i>)</code>
Parameters	<p><i>filename</i></p> <p>A file containing the macros to be registered (string). The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide</i>.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.
Example	<code>__registerMacroFile("c:\\testdir\\macro.mac");</code>
See also	<i>Using C-SPY macros</i> , page 209.

## \_\_resetFile

Syntax	<code>__resetFile(<i>fileHandle</i>)</code>
Parameters	<p><i>fileHandle</i></p> <p>A macro variable used as filehandle by the <code>__openFile</code> macro.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Rewinds a file previously opened by <code>__openFile</code> .

## \_\_setCodeBreak

Syntax	<code>__setCodeBreak(location, count, condition, cond_type, action)</code>						
Parameters	<p><i>location</i></p> <p>A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see <i>Enter Location dialog box</i>, page 138.</p> <p><i>count</i></p> <p>The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).</p> <p><i>condition</i></p> <p>The breakpoint condition (string).</p> <p><i>cond_type</i></p> <p>The condition type; either "CHANGED" or "TRUE" (string).</p> <p><i>action</i></p> <p>An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.</p>						
Return value	<table border="1"> <thead> <tr> <th>Result</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Successful</td> <td>An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.</td> </tr> <tr> <td>Unsuccessful</td> <td>0</td> </tr> </tbody> </table> <p><i>Table 25: __setCodeBreak return values</i></p>	Result	Value	Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.	Unsuccessful	0
Result	Value						
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.						
Unsuccessful	0						
For use with	All C-SPY drivers.						
Description	Sets a code breakpoint, that is, a breakpoint which is triggered just before the processor fetches an instruction at the specified location.						
Examples	<pre>__setCodeBreak("{D:\\src\\prog.c}.12.9", 3, "d&gt;16", "TRUE", "ActionCode()");</pre> <p>This example sets a code breakpoint on the label <code>main</code> in your source:</p> <pre>__setCodeBreak("main", 0, "1", "TRUE", "");</pre>						
See also	<i>Breakpoints</i> , page 113.						

## \_\_setComplexBreak

### Syntax

```
__setComplexBreak(control, a_addr, b_addr, access_type,
a_access, b_access, complex_data, c_value, d_value, c_compare,
d_compare, action)
```

### Parameters

<i>control</i>	<p>Breakpoint control:</p> <p>ENABLE_A to enable a breakpoint at <i>a_addr</i></p> <p>ENABLE_AB to enable breakpoints at <i>a_addr</i> and <i>b_addr</i></p> <p>RANGE to enable a range breakpoint from <i>a_addr</i> to <i>b_addr</i>.</p>
<i>a_addr</i>	<p>A string with a location description. This can be:</p> <p>A source location on the form <i>{filename}.line.col</i>, for example <i>{D:\\src\\prog.c}.12.9</i>, although this is not very useful for data breakpoints.</p> <p>An absolute location on the form <i>zone:hexaddress</i> or simply <i>hexaddress</i>, for example <i>Memory:0x42</i></p> <p>An expression whose value designates a location, for example <i>myGlobalVariable</i>.</p>
<i>b_addr</i>	<p>A string with a location description. This can be:</p> <p>A source location on the form <i>{filename}.line.col</i>, for example <i>{D:\\src\\prog.c}.12.9</i>, although this is not very useful for data breakpoints.</p> <p>An absolute location on the form <i>zone:hexaddress</i> or simply <i>hexaddress</i>, for example <i>Memory:0x42</i>.</p> <p>An expression whose value designates a location, for example <i>myGlobalVariable</i>.</p>
<i>access_type</i>	<p>The memory space:</p> <p>DATA for data memory</p> <p>CODE for code memory</p>
<i>a_access</i>	<p>The memory access type:</p> <p>R for read</p> <p>W for write</p> <p>RW for read/write</p>

<i>b_access</i>	<p>The memory access type:</p> <p>R for read</p> <p>W for write</p> <p>RW for read/write</p>
<i>complex_data</i>	<p>Complex data control:</p> <p>C_COMBINED_WITH_A for enable complex data</p> <p>CD_COMBINED_WITH_AB for C combined with A and D combined with B</p> <p>CD_COMBINED_WITH_A for C and D combined with A</p> <p>NOTCD_COMBINED_WITH_A for not C and D combined with A</p> <p>C_MASKED_WITH_D_COMBINED_WITH_A for C masked with D combined with A</p>
<i>c_value</i>	<p>Single-byte value for comparison with the memory contents of <i>a_addr</i> and <i>b_addr</i>.</p>
<i>d_value</i>	<p>Single-byte value for comparison with the memory contents of <i>a_addr</i> and <i>b_addr</i>.</p>
<i>c_compare</i>	<p>The relationship between <i>c_value</i> and the contents of data memory at <i>a_addr</i> and <i>b_addr</i>:</p> <p>EQ matches when <i>c_value</i> and the data value are equal</p> <p>LE matches when <i>c_value</i> is less than the data value</p> <p>GE matches when <i>c_value</i> is greater than or equal to the data value</p>
<i>d_compare</i>	<p>The relationship between <i>d_value</i> and the contents of data memory at <i>a_addr</i> and <i>b_addr</i>:</p> <p>EQ matches when <i>d_value</i> and the data value are equal</p> <p>LE matches when <i>d_value</i> is less than the data value</p> <p>GE matches when <i>d_value</i> is greater than or equal to the data value</p>
<i>action</i>	<p>An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.</p>

## Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 26: `__set` Complex Break return values

## For use with

The C-SPY AVR ONE! driver  
 The C-SPY Atmel-ICE driver  
 The C-SPY JTAGICE3 driver

## Description

Sets a complex breakpoint, that is, a breakpoint which is triggered directly after the processor has read or written data at the specified location.

## Example

The following example enables one data read/write breakpoint at the address for variable `a` and will break if the memory byte value at address is equal to `0x22`, and it enables one data read breakpoint at the address for variable `b` and will break if the memory byte value at address is greater than or equal to `0x11`. When a break occurs, the macro function `ActionData()` will be called.

```
__var brk;
brk=__setComplexBreak("ENABLE_AB", "a", "b", "DATA", "RW", "R",
"CD_COMBINED_WITH_AB", "0x22", "0x11" "EQ", "GE",
"ActionData()");
...
__clearBreak(brk);
```

## See also

*Breakpoints*, page 113.

**`__setDataBreak`**

## Syntax

```
__setDataBreak(location, count, condition, cond_type, access,
action)
```

## Parameters

*location*

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see Enter Location dialog box, page 25.

*count*

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).

*condition*

The breakpoint condition (string).

*cond\_type*

The condition type; either "CHANGED" or "TRUE" (string).

*access*

The memory access type: "R", for read, "W" for write, or "RW" for read/write.

*action*

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 27: *\_\_setDataBreak* return values

For use with

The C-SPY Simulator  
 The C-SPY Atmel-ICE driver  
 The C-SPY JTAGICE3 driver  
 The C-SPY AVR ONE! driver

Description

Sets a data breakpoint, that is, a breakpoint which is triggered directly after the processor has read or written data at the specified location.

Example

```
__var brk;
brk = __setDataBreak("DATA:0x4710", 3, "d>6", "TRUE",
    "W", "ActionData()");
...
__clearBreak(brk);
```

See also

*Breakpoints*, page 113.

## \_\_setLogBreak

### Syntax

```
__setLogBreak(location, message, msg_type, condition,
             cond_type)
```

### Parameters

*location*

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box*, page 138.

*message*

The message text.

*msg\_type*

The message type; choose between:

TEXT, the message is written word for word.

ARGS, the message is interpreted as a comma-separated list of C-SPY expressions or strings.

*condition*

The breakpoint condition (string).

*cond\_type*

The condition type; either "CHANGED" or "TRUE" (string).

### Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	0

Table 28: \_\_setLogBreak return values

### For use with

All C-SPY drivers.

### Description

Sets a log breakpoint, that is, a breakpoint which is triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily halt and print the specified message in the C-SPY **Debug Log** window.

**Example**

```

__var logBp1;
__var logBp2;

logOn()
{
    logBp1 = __setLogBreak ("{C:\\temp\\Utilities.c}.23.1",
        "\\Entering trace zone at :\\", #PC:%X", "ARGS", "1", "TRUE");
    logBp2 = __setLogBreak ("{C:\\temp\\Utilities.c}.30.1",
        "Leaving trace zone...", "TEXT", "1", "TRUE");
}

logOff()
{
    __clearBreak(logBp1);
    __clearBreak(logBp2);
}

```

**See also**

*Formatted output*, page 217 and *Breakpoints*, page 113.

**\_\_setSimBreak****Syntax**

```
__setSimBreak(location, access, action)
```

**Parameters**

*location*

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see *Enter Location dialog box*, page 138.

*access*

The memory access type: "R" for read or "W" for write.

*action*

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

**Return value**

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 29: \_\_setSimBreak return values

**For use with**

The C-SPY Simulator.



**Description** Use this system macro to set *immediate* breakpoints, which will halt instruction execution only temporarily. This allows a C-SPY macro function to be called when the processor is about to read data from a location or immediately after it has written data. Instruction execution will resume after the action.

This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the processor reads at a memory-mapped location, a C-SPY macro function can intervene and supply the appropriate data. Conversely, when the processor writes to a memory-mapped location, a C-SPY macro function can act on the value that was written.

## \_\_setTraceStartBreak

**Syntax** `__setTraceStartBreak(location)`

**Parameters** *location*

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box*, page 138.

**Return value**

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	0

Table 30: \_\_setTraceStartBreak return values

**For use with** The C-SPY Simulator.

**Description** Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is started.

## Example

```

__var startTraceBp;
__var stopTraceBp;

traceOn()
{
    startTraceBp = __setTraceStartBreak
        ("C:\\TEMP\\Utilities.c).23.1");
    stopTraceBp = __setTraceStopBreak
        ("C:\\temp\\Utilities.c).30.1");
}

traceOff()
{
    __clearBreak(startTraceBp);
    __clearBreak(stopTraceBp);
}

```

## See also

*Breakpoints*, page 113.

## \_\_setTraceStopBreak

## Syntax

```
__setTraceStopBreak(location)
```

## Parameters

*location*

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box*, page 138.

## Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	int 0

Table 31: \_\_setTraceStopBreak return values

## For use with

The C-SPY Simulator.

## Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is stopped.

## Example

See *\_\_setTraceStartBreak*, page 249.

See also *Breakpoints*, page 113.

## \_\_sourcePosition

Syntax `__sourcePosition(linePtr, colPtr)`

Parameters

*linePtr*  
Pointer to the variable storing the line number

*colPtr*  
Pointer to the variable storing the column number

Return value

Result	Value
Successful	Filename string
Unsuccessful	Empty ( " " ) string

Table 32: *\_\_sourcePosition* return values

For use with All C-SPY drivers.

Description If the current execution location corresponds to a source location, this macro returns the filename as a string. It also sets the value of the variables, pointed to by the parameters, to the line and column numbers of the source location.

## \_\_strFind

Syntax `__strFind(macroString, pattern, position)`

Parameters

*macroString*  
A macro string.

*pattern*  
The string pattern to search for

*position*  
The position where to start the search. The first position is 0

Return value The position where the pattern was found or -1 if the string is not found.

For use with All C-SPY drivers.

Description	This macro searches a given string ( <i>macroString</i> ) for the occurrence of another string ( <i>pattern</i> ).
Example	<pre>__strFind("Compiler", "pile", 0) = 3 __strFind("Compiler", "foo", 0) = -1</pre>
See also	<i>Macro strings</i> , page 215.

## **\_\_subString**

Syntax	<code>__subString(<i>macroString</i>, <i>position</i>, <i>length</i>)</code>
Parameters	<p><i>macroString</i> A macro string.</p> <p><i>position</i> The start position of the substring. The first position is 0.</p> <p><i>length</i> The length of the substring</p>
Return value	A substring extracted from the given macro string.
For use with	All C-SPY drivers.
Description	This macro extracts a substring from another string ( <i>macroString</i> ).
Example	<pre>__subString("Compiler", 0, 2) The resulting macro string contains Co. __subString("Compiler", 3, 4) The resulting macro string contains pile.</pre>
See also	<i>Macro strings</i> , page 215.

## **\_\_targetDebuggerVersion**

Syntax	<code>__targetDebuggerVersion()</code>
Return value	A string that represents the version number of the C-SPY debugger processor module.
For use with	All C-SPY drivers.

**Description** This macro returns the version number of the C-SPY debugger processor module.

**Example**

```
__var toolVer;
toolVer = __targetDebuggerVersion();
__message "The target debugger version is, ", toolVer;
```

## \_\_toLower

**Syntax** `__toLower(macroString)`

**Parameters** *macroString*

A macro string.

**Return value** The converted macro string.

**For use with** All C-SPY drivers.

**Description** This macro returns a copy of the parameter *macroString* where all the characters have been converted to lower case.

**Example**

```
__toLower("IAR")
The resulting macro string contains iar.
__toLower("Mix42")
The resulting macro string contains mix42.
```

**See also** *Macro strings*, page 215.

## \_\_toString

**Syntax** `__toString(C_string, maxlength)`

**Parameters** *C\_string*

Any null-terminated C string.

*maxlength*

The maximum length of the returned macro string.

**Return value** Macro string.

**For use with** All C-SPY drivers.

Description	This macro is used for converting C strings ( <code>char*</code> or <code>char []</code> ) into macro strings.
Example	Assuming your application contains this definition: <pre>char const * hptr = "Hello World!";</pre> this macro call: <pre>__toString(hptr, 5)</pre> would return the macro string containing <code>Hello</code> .
See also	<i>Macro strings</i> , page 215.

## **\_\_ToUpper**

Syntax	<code>__ToUpper(<i>macroString</i>)</code>
Parameters	<i>macroString</i> A macro string.
Return value	The converted string.
For use with	All C-SPY drivers.
Description	This macro returns a copy of the parameter <i>macroString</i> where all the characters have been converted to upper case.
Example	<pre>__ToUpper("string")</pre> The resulting macro string contains <code>STRING</code> .
See also	<i>Macro strings</i> , page 215.

## **\_\_transparent**

Syntax	<code>__transparent(<i>commandstring</i>)</code>
Parameters	<i>commandstring</i> The command to send to the ROM-monitor.
For use with	The C-SPY CCR driver.

**Description** Sends a transparent command directly to the ROM-monitor. In this way, you can communicate directly with the ROM-monitor transparently from C-SPY.

**See also** *Using C-SPY macros for transparent commands*, page 343.

## \_\_unloadImage

**Syntax** `__unloadImage(module_id)`

**Parameters** *module\_id*  
An integer which represents a unique module identification, which is retrieved as a return value from the corresponding `__loadImage` C-SPY macro.

**Return value**

Value	Result
<i>module_id</i>	A unique module identification (the same as the input parameter).
int 0	The unloading failed.

*Table 33: \_\_unloadImage return values*

**For use with** All C-SPY drivers.

**Description** Unloads debug information from an already downloaded image.

**See also** *Loading multiple images*, page 55 and *Images*, page 296.

## \_\_writeFile

**Syntax** `__writeFile(fileHandle, value)`

**Parameters** *fileHandle*  
A macro variable used as filehandle by the `__openFile` macro.  
*value*  
An integer.

**Return value** int 0

**For use with** All C-SPY drivers.

**Description** Prints the integer value in hexadecimal format (with a trailing space) to the file *file*.

**Note:** The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.

## **\_\_writeFileByte**

Syntax	<code>__writeFileByte(<i>fileHandle</i>, <i>value</i>)</code>
Parameters	<p><i>fileHandle</i></p> <p>A macro variable used as filehandle by the <code>__openFile</code> macro.</p> <p><i>value</i></p> <p>An integer.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes one byte to the file <i>fileHandle</i> .

## **\_\_writeMemory8, \_\_writeMemoryByte**

Syntax	<code>__writeMemory8(<i>value</i>, <i>address</i>, <i>zone</i>)</code> <code>__writeMemoryByte(<i>value</i>, <i>address</i>, <i>zone</i>)</code>
Parameters	<p><i>value</i></p> <p>An integer.</p> <p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 142.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes one byte to a given memory location.
Example	<code>__writeMemory8(0x2F, 0x8020, "DATA");</code>



## \_\_writeMemory16

Syntax	<code>__writeMemory16(value, address, zone)</code>
Parameters	<p><i>value</i> An integer.</p> <p><i>address</i> The memory address (integer).</p> <p><i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 142.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes two bytes to a given memory location.
Example	<code>__writeMemory16(0x2FFF, 0x8020, "DATA");</code>

## \_\_writeMemory32

Syntax	<code>__writeMemory32(value, address, zone)</code>
Parameters	<p><i>value</i> An integer.</p> <p><i>address</i> The memory address (integer).</p> <p><i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 142.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes four bytes to a given memory location.
Example	<code>__writeMemory32(0x5555FFFF, 0x8020, "DATA");</code>

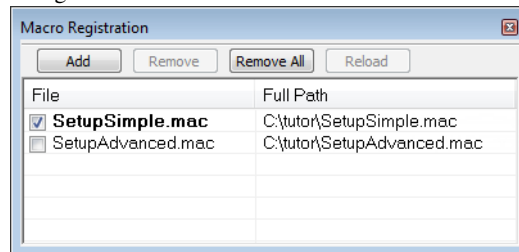
## Graphical environment for macros

Reference information about:

- *Macro Registration window*, page 258
- *Debugger Macros window*, page 260
- *Macro Quicklaunch window*, page 262

### Macro Registration window

The Macro Registration window is available from the **View>Macros** submenu during a debug session.



Use this window to list, register, and edit your debugger macro files.

Double-click a macro file to open it in the editor window and edit it.

#### Requirements

None; this window is always available.

#### Display area

This area contains these columns:

##### File

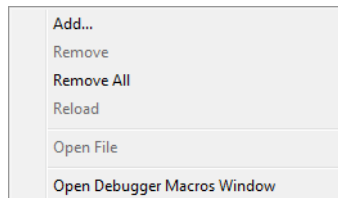
The name of an available macro file. To register the macro file, select the check box to the left of the filename. The name of a registered macro file appears in bold style.

##### Full path

The path to the location of the added macro file.

## Context menu

This context menu is available:



These commands are available:

### **Add**

Opens a file browser where you can locate the macro file that you want to add to the list. This menu command is also available as a function button at the top of the window.

### **Remove**

Removes the selected debugger macro file from the list. This menu command is also available as a function button at the top of the window.

### **Remove All**

Removes all macro files from the list. This menu command is also available as a function button at the top of the window.

### **Reload**

Registers the selected macro file. Typically, this is useful when you have edited a macro file. This menu command is also available as a function button at the top of the window.

### **Open File**

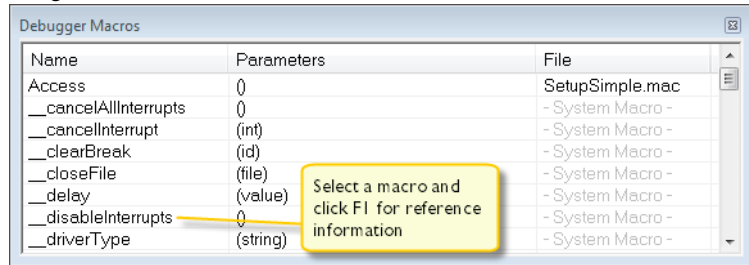
Opens the selected macro file in the editor window.

### **Open Debugger Macros Window**

Opens the Debugger Macros window.

## Debugger Macros window

The Debugger Macros window is available from the **View>Macro** submenu during a debug session.



Use this window to list all registered debugger macro functions, either predefined system macros or your own. This window is useful when you edit your own macro functions and want an overview of all available macros that you can use.

Double-clicking a macro defined in a file opens that file in the editor window.

To open a macro in the **Macro Quicklaunch** window, drag it from the **Debugger Macros** window and drop it in the **Macro Quicklaunch** window.

Select a macro and press F1 to get online help information for that macro.

### Requirements

None; this window is always available.

### Display area

This area contains these columns:

#### Name

The name of the debugger macro.

#### Parameters

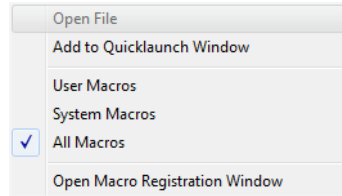
The parameters of the debugger macro.

#### File

For macros defined in a file, the name of the file is displayed. For predefined system macros, -System Macro- is displayed.

## Context menu

This context menu is available:



These commands are available:

### **Open File**

Opens the selected debugger macro file in the editor window.

### **Add to Quicklaunch Window**

Adds the selected macro to the **Macro Quicklaunch** window.

### **User Macros**

Lists only the debugger macros that you have defined yourself.

### **System Macros**

Lists only the predefined system macros.

### **All Macros**

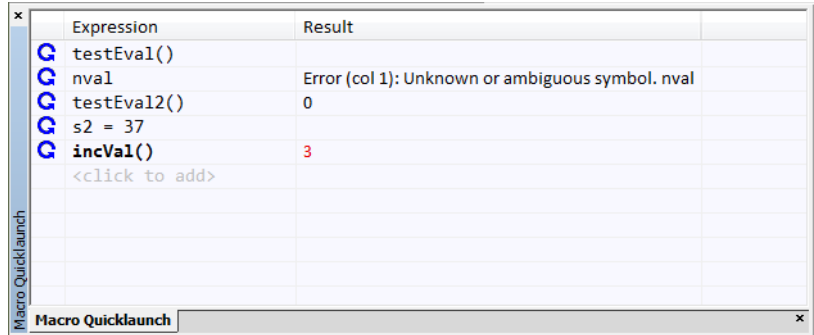
Lists all debugger macros, both predefined system macros and your own.

### **Open Macro Registration Window**

Opens the **Macro Registration** window.

## Macro Quicklaunch window

The **Macro Quicklaunch** window is available from the **View** menu.



Use this window to evaluate expressions, typically C-SPY macros.

For some devices, there are predefined C-SPY macros available with device support, typically provided by the chip manufacturer. These macros are useful for performing certain device-specific tasks. The macros are available in the **Macro Quicklaunch** window and are easily identified by their green icon,


The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is primarily designed for evaluating C-SPY macros. The window gives you precise control over when to evaluate an expression.

### To add an expression:

- I Choose one of these alternatives:
  - Drag the expression to the window
  - In the **Expression** column, type the expression you want to examine.

If the expression you add and want to evaluate is a C-SPY macro, the macro must first be registered, see *Using C-SPY macros*, page 209.

### To evaluate an expression:

- I  Double-click the **Recalculate** icon to calculate the value of that expression.

### Requirements

None; this window is always available.

## Display area

This area contains these columns:



### Recalculate icon

To evaluate the expression, double-click the icon. The latest evaluated expression appears in bold style.

### Expression

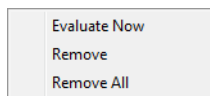
One or several expressions that you want to evaluate. Click <click to add> to add an expression. If the return value has changed since last time, the value will be displayed in red.

### Result

Shows the return value from the expression evaluation.

## Context menu

This context menu is available:



These commands are available:

### Evaluate Now

Evaluates the selected expression.

### Remove

Removes the selected expression.

### Remove All

Removes all selected expressions.





# The C-SPY command line utility—`cspybat`

- Using C-SPY in batch mode
- Summary of C-SPY command line options
- Reference information on C-SPY command line options.

---

## Using C-SPY in batch mode

You can execute C-SPY in batch mode if you use the command line utility `cspybat`, installed in the directory `common\bin`.

These topics are covered:

- Starting `cspybat`
- Output
- Invocation syntax

### STARTING CSPYBAT

- 1 To start `cspybat` you must first create a batch file. An easy way to do that is to use one of the batch files that C-SPY automatically generates when you start C-SPY in the IDE.

C-SPY generates a batch file `projectname.buildconfiguration.cspy.bat` every time C-SPY is initialized. In addition, two more files are generated:

- `project.buildconfiguration.general.xml`, which contains options specific to `cspybat`.
- `project.buildconfiguration.driver.xml`, which contains options specific to the C-SPY driver you are using.

You can find the files in the directory `$PROJ_DIR$\settings`. The files contain the same settings as the IDE, and provide hints about additional options that you can use.

- 2 To start `cspybat`, you can use this command line:

```
project.cspybat.bat [debugfile]
```

Note that *debugfile* is optional. You can specify it if you want to use a different debug file than the one that is used in the *project.buildconfiguration.general.xml* file.

## OUTPUT

When you run `cspybat`, these types of output can be produced:

- Terminal output from `cspybat` itself
 

All such terminal output is directed to `stderr`. Note that if you run `cspybat` from the command line without any arguments, the `cspybat` version number and all available options including brief descriptions are directed to `stdout` and displayed on your screen.
- Terminal output from the application you are debugging
 

All such terminal output is directed to `stdout`, provided that you have used the `--plugin` option. See *--plugin*, page 287.
- Error return codes
 

`cspybat` returns status information to the host operating system that can be tested in a batch file. For *successful*, the value `int 0` is returned, and for *unsuccessful* the value `int 1` is returned.

## INVOCATION SYNTAX

The invocation syntax for `cspybat` is:

```
cspybat processor_DLL driver_DLL debug_file
        [cspybat_options] --backend driver_options
```

**Note:** In those cases where a filename is required—including the DLL files—you are recommended to give a full path to the filename.

## Parameters

The parameters are:

Parameter	Description
<i>processor_DLL</i>	The processor-specific DLL file; available in <code>avr\bin</code> .
<i>driver_DLL</i>	The C-SPY driver DLL file; available in <code>avr\bin</code> .
<i>debug_file</i>	The object file that you want to debug (filename extension <code>d90</code> ). See also <i>-debugfile</i> , page 274.
<i>cspybat_options</i>	The command line options that you want to pass to <code>cspybat</code> . Note that these options are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 271.

Table 34: *cspybat* parameters

Parameter	Description
<code>--backend</code>	Marks the beginning of the parameters to the C-SPY driver; all options that follow will be sent to the driver. Note that this option is mandatory.
<code>driver_options</code>	The command line options that you want to pass to the C-SPY driver. Note that some of these options are mandatory and some are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 271.

Table 34: cspybat parameters (Continued)

## Summary of C-SPY command line options

Reference information about:

- General cspybat options
- Options available for all C-SPY drivers
- Options available for the simulator driver
- Options available for all C-SPY hardware debugger drivers
- Options available for the C-SPY Atmel-ICE driver
- Options available for the C-SPY JTAGICE driver, the C-SPY JTAGICE mkII driver, the C-SPY Dragon driver, the C-SPY JTAGICE driver, and the C-SPY AVR ONE! driver
- Options available for the C-SPY JTAGICE mkII driver, the C-SPY Dragon driver, the C-SPY Atmel-ICE driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver
- Options available for the C-SPY JTAGICE driver, the C-SPY JTAGICE mkII driver, and the C-SPY Dragon driver
- Options available for the C-SPY Atmel-ICE driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver
- Options available for the C-SPY JTAGICE mkII driver and the C-SPY Dragon driver
- Options available for the C-SPY Dragon driver
- Options available for the C-SPY ICE200 driver

### GENERAL CSPYBAT OPTIONS

`--backend` Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).

<code>--code_coverage_file</code>	Enables the generation of code coverage information and places it in a specified file.
<code>--cycles</code>	Specifies the maximum number of cycles to run.
<code>--debugfile</code>	Specifies an alternative debug file.
<code>--download_only</code>	Downloads a code image without starting a debug session afterwards.
<code>-f</code>	Extends the command line.
<code>--leave_running</code>	Starts the execution on the target and then exits but leaves the target running.
<code>--macro</code>	Specifies a macro file to be used.
<code>--macro_param</code>	Assigns a value to a C-SPY macro parameter.
<code>--plugin</code>	Specifies a plugin file to be used.
<code>--silent</code>	Omits the sign-on message.
<code>--timeout</code>	Limits the maximum allowed execution time.

### OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

<code>--64bit_doubles</code>	Specifies that 64-bit doubles are used instead of 32-bit doubles.
<code>--64k_flash</code>	Enables 64-Kbytes flash mode for the processor configurations <code>-v2</code> and <code>-v3</code> .
<code>--cpu</code>	Specifies the CPU model your application was compiled for.
<code>-d</code>	Specifies the C-SPY driver to be used.
<code>--disable_internal_eeprom</code>	Disables the internal EEPROM.
<code>--eeprom_size</code>	Specifies the size of the built-in EEPROM area.
<code>--enhanced_core</code>	Enables the enhanced instruction set.
<code>-p</code>	Specifies the device description file to be used.
<code>-v</code>	Specifies the processor configuration your application was compiled for.

## OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

`--disable_interrupts` Disables the interrupt simulation.

## OPTIONS AVAILABLE FOR ALL C-SPY HARDWARE DEBUGGER DRIVERS

`--drv_communication` Specifies the communication channel to be used between C-SPY and the target system.

`--drv_communication_log` Logs the communication between C-SPY and the target system.

`--drv_download_data` Enables downloading of constant data into RAM.

`--drv_suppress_download` Suppresses download of the executable image.

`--drv_verify_download` Verifies the executable image.

## OPTIONS AVAILABLE FOR THE C-SPY ATMEL-ICE DRIVER

`--drv_atmel_ice` Specifies the C-SPY Atmel-ICE driver to be used.

## OPTIONS AVAILABLE FOR THE C-SPY JTAGICE DRIVER, THE C-SPY JTAGICE MKII DRIVER, THE C-SPY DRAGON DRIVER, THE C-SPY JTAGICE DRIVER, AND THE C-SPY AVR ONE! DRIVER

`--drv_set_exit_breakpoint` Sets a system breakpoint on the `exit` label.

`--drv_set_getchar_breakpoint` Sets a system breakpoint on the `getchar` label.

`--drv_set_putchar_breakpoint` Sets a system breakpoint on the `putchar` label.

`--jtagice_do_hardware_reset` Makes the hardware reset every time the debugger is reset.

`--jtagice_leave_timers_running` Ensures that the timers always run, even if the application is stopped.

`--jtagice_preserve_eeprom` Preserves the EEPROM contents even if the device is reprogrammed.

`--jtagice_restore_fuse` Allows the debugger to modify the OCD enable fuse and preserve the EEPROM fuse at startup.

**OPTIONS AVAILABLE FOR THE C-SPY JTAGICE MKII DRIVER, THE C-SPY DRAGON DRIVER, THE C-SPY ATMEL-ICE DRIVER, THE C-SPY JTAGICE3 DRIVER, AND THE C-SPY AVR ONE! DRIVER**

`--drv_preserve_app_section` Preserves the application area of the flash memory during download.

`--drv_preserve_boot_section` Preserves the boot area of the flash memory during download.

**OPTIONS AVAILABLE FOR THE C-SPY JTAGICE DRIVER, THE C-SPY JTAGICE MKII DRIVER, AND THE C-SPY DRAGON DRIVER**

`--jtagice_clock` Specifies the speed of the JTAG clock.

**OPTIONS AVAILABLE FOR THE C-SPY ATMEL-ICE DRIVER, THE C-SPY JTAGICE3 DRIVER, AND THE C-SPY AVR ONE! DRIVER**

`--avrone_jtag_clock` Specifies the speed of the debugging interface.

`--drv_debug_port` Specifies the debug interface.

**OPTIONS AVAILABLE FOR THE C-SPY JTAGICE MKII DRIVER AND THE C-SPY DRAGON DRIVER**

`--drv_use_PDI` Makes the C-SPY driver communicate with the device using the PDI interface.

`--jtagicemkII_use_software_brea  
kpoints` Makes software breakpoints available.

**OPTIONS AVAILABLE FOR THE C-SPY DRAGON DRIVER**

`--drv_dragon` Specifies the AVR Dragon driver to be used.

**OPTIONS AVAILABLE FOR THE C-SPY ICE200 DRIVER**

`--ice200_restore_EEPROM` Restores the contents of the on-chip EEPROM data memory.

`--ice200_single_step_timers` Allows the timers to single-step.

---

**Reference information on C-SPY command line options**

This section gives detailed reference information about each `cspybat` option and each option available to the C-SPY drivers.

**--64bit\_doubles**

Syntax `--64bit_doubles`

For use with All C-SPY drivers.

Description Use this option to specify that 64-bit doubles are used instead of 32-bit doubles.



**Project>Options>General Options>Target>Use 64-bit doubles**

**--64k\_flash**

Syntax `--64k_flash`


For use with All C-SPY drivers.

Description Use this option to enable 64-Kbytes flash mode for the processor configurations `-v2` and `-v3`.




**Project>Options>General Options>Target>No RAMPZ register**

**--avrone\_jtag\_clock**

Syntax	<code>--avrone_jtag_clock=<i>speed</i></code>
Parameters	<i>speed</i> The JTAG or PDI clock frequency in Hz. Possible values are 0-65535000 Hz in steps of 1000 Hz.
For use with	The C-SPY Atmel-ICE driver. The C-SPY AVR ONE! driver The C-SPY JTAGICE3 driver
Description	Use this option to specify the speed of the debugging interface.  <b>Project&gt;Options&gt;Debugger&gt;Driver&gt;Driver 1&gt;Debug Port&gt;Frequency in kHz</b>


**--backend**

Syntax	<code>--backend {<i>driver options</i>}</code>
Parameters	<i>driver options</i> Any option available to the C-SPY driver you are using.
For use with	<code>cspybat</code> (mandatory).
Description	Use this option to send options to the C-SPY driver. All options that follow <code>--backend</code> will be passed to the C-SPY driver, and will not be processed by <code>cspybat</code> itself.  This option is not available in the IDE.


**--code\_coverage\_file**

Syntax	<code>--code_coverage_file <i>file</i></code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>file</i> The name of the destination file for the code coverage information.



For use with	<code>cspybat</code>
Description	<p>Use this option to enable the generation of code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file.</p> <p>Note that this option requires that the C-SPY driver you are using supports code coverage. If you try to use this option with a C-SPY driver that does not support code coverage, an error message will be directed to <code>stderr</code>.</p>
See also	<p><i>Code coverage</i>, page 193.</p> <p> To set this option, choose <b>View&gt;Code Coverage</b>, right-click and choose <b>Save As</b> when the C-SPY debugger is running.</p>

## --cpu

Syntax	<code>--cpu=<i>cpu_name</i></code>
Parameters	<p><i>cpu_name</i>                      The CPU model, xm128a1, m2560, tiny441, etc.</p>
For use with	All C-SPY drivers.
Description	<p>Use this option to specify the CPU model your application was compiled for. This option cannot be used together with <code>-v</code>.</p> <p> <b>Project&gt;Options&gt;General Options&gt;Target&gt;Processor configuration</b></p>

## --cycles

Syntax	<code>--cycles <i>cycles</i></code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<p><i>cycles</i></p> <p>The number of cycles to run.</p>
For use with	<code>cspybat</code>
Description	Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted.

Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.



This option is not available in the IDE.

## -d

Syntax `-d {driver|thirdpartydriver}`

### Parameters

*driver* Specifies the C-SPY driver. you are using. Choose between:

- `avrone`
- `ccr`
- `ice200`
- `jtagice`
- `jtagicemkII`
- `jtagice3`
- `sim`

For use with All C-SPY drivers.

Description Use this option to specify the C-SPY driver to be used.



**Project>Options>Debugger>Setup>Driver**

## --debugfile

Syntax `--debugfile filename`

### Parameters

*filename*  
The name of the debug file to use.

For use with `cspybat`

This option can be placed both before and after the `--backend` option on the command line.

**Description** Use this option to make `cspybat` use the specified debugfile instead of the one used in the generated `cpsybat.bat` file.



This option is not available in the IDE.

## --disable\_internal\_eeprom

**Syntax** `--disable_internal_eeprom`

**For use with** All C-SPY drivers.

**Description** Use this option to disable the internal EEPROM.



To set related options, choose:

**Project>Options>General Options>Target>Utilize inbuilt EEPROM**

## --disable\_interrupts

**Syntax** `--disable_interrupts`

**For use with** The C-SPY Simulator driver.

**Description** Use this option to disable the interrupt simulation.



To set this option, choose **Simulator>Interrupt Setup** and deselect the **Enable interrupt simulation** option.

## --download\_only

**Syntax** `--download_only`

Note that this option must be placed before the `--backend` option on the command line.

**For use with** `cspybat`

**Description** Use this option to download the code image without starting a debug session afterwards.



To set related option, choose:

**Project>Options>Debugger>Setup** and deselect **Run to**.

## --drv\_atmel\_ice

Syntax	--drv_atmel_ice
For use with	The C-SPY Atmel-ICE driver.
Description	Use this option to specify the C-SPY Atmel-ICE driver to be used.



**Project>Options>Debugger>Driver**

## --drv\_communication

Syntax	--drv_communication=[COM $n$  USB]
Parameters	<p>COM<math>n</math>      A serial communication port. <math>n</math> can be between 1 and 32. Note that COM<math>n</math> is not used in AVR ONE! and JTAGICE3.</p> <p>USB         The USB port. Can only be used by the AVR ONE!, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.</p>
For use with	All C-SPY hardware drivers.
Description	Use this option to specify the communication channel to be used between C-SPY and the target system.



**Project>Options>Debugger>Driver**

## --drv\_communication\_log

Syntax	--drv_communication_log= <i>filename</i>
Parameters	<p><i>filename</i>                      The name of the log file.</p>
For use with	All C-SPY hardware drivers.
Description	Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the communication protocol is required.



```

Project>Options>Debugger>Atmel-ICE>Communication>Log communication
Project>Options>Debugger>AVR ONE!>Communication>Log communication
Project>Options>Debugger>CCR>Serial Port>Log communication
Project>Options>Debugger>ICE200>Serial Port>Log communication
Project>Options>Debugger>JTAGICE>Serial Port>Log communication
Project>Options>Debugger>JTAGICE3>Communication>Log communication
Project>Options>Debugger>JTAGICE mkII>Serial Port>Log communication
Project>Options>Debugger>Dragon>Communication>Log communication

```

## --drv\_debug\_port

Syntax	<code>--drv_debug_port={autodetect debugwire pdi jtag}</code>	
Parameters		
	<code>autodetect</code>	Specifies auto-detection of the debugging interface.
	<code>debugwire</code>	Specifies the debugWIRE debugging interface.
	<code>pdi</code>	Specifies the PDI debugging interface.
	<code>jtag</code>	Specifies the JTAG debugging interface.
For use with	The C-SPY Atmel-ICE driver The C-SPY AVR ONE! driver The C-SPY JTAGICE3 driver	
Description	Use this option to specify the debug interface.	



```

Project>Options>Debugger>Atmel-ICE>Atmel-ICE 1>Debug Port
Project>Options>Debugger>AVR ONE!>AVR ONE! 1>Debug Port
Project>Options>Debugger>JTAGICE3>JTAGICE3 1>Debug Port

```

## --drv\_download\_data

Syntax	<code>--drv_download_data</code>
For use with	All C-SPY hardware drivers.

Description Use this option to enable downloading of constant data into RAM.



**Project>Options>Debugger>Driver>Driver 1>Allow download to RAM**

## **--drv\_dragon**

Syntax `--drv_dragon`

For use with The C-SPY AVR Dragon driver.

Description Use this option to specify the AVR Dragon driver to be used.



**Project>Options>Debugger>Driver**

## **--drv\_preserve\_app\_section**

Syntax `--drv_preserve_app_section`

For use with  
 The C-SPY Atmel-ICE driver  
 The C-SPY AVR ONE! driver  
 The C-SPY AVR Dragon driver  
 The C-SPY JTAGICE mkII driver  
 The C-SPY JTAGICE3 driver

Description Use this option to preserve the application area of the flash memory during download.




**Project>Options>Debugger>Driver>Driver 2>Preserve FLASH>Application Area**


## **--drv\_preserve\_boot\_section**

Syntax `--drv_preserve_boot_section`

For use with  
 The C-SPY Atmel-ICE driver  
 The C-SPY AVR ONE! driver  
 The C-SPY JTAGICE3 driver

	The C-SPY AVR Dragon driver
	The C-SPY JTAGICE mkII driver
Description	Use this option to preserve the boot area of the flash memory during download.
	 <b>Project&gt;Options&gt;Debugger&gt;Driver&gt;Driver 2&gt;Preserve FLASH&gt;Boot Area</b>

## --drv\_set\_exit\_breakpoint

Syntax	--drv_set_exit_breakpoint
For use with	The C-SPY Atmel-ICE driver The C-SPY AVR ONE! driver The C-SPY AVR Dragon driver The C-SPY JTAGICE driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option in the CLIB runtime environment to set a system breakpoint on the <code>exit</code> label. This consumes a hardware breakpoint.
See also	<i>Breakpoint consumers</i> , page 118
	 <b>Project&gt;Options&gt;Debugger&gt;Driver&gt;Driver 2&gt;System breakpoints on&gt;exit</b>

## --drv\_set\_getchar\_breakpoint

Syntax	--drv_set_getchar_breakpoint
For use with	The C-SPY Atmel-ICE driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver

**Description** Use this option in the CLIB runtime environment to set a system breakpoint on the `getchar` label. This consumes a hardware breakpoint.

**See also** *Breakpoint consumers*, page 118



**Project>Options>Debugger>Driver>Driver 2>System breakpoints on>getchar**

## **--drv\_set\_putchar\_breakpoint**

**Syntax** `--drv_set_putchar_breakpoint`

**For use with**

- The C-SPY Atmel-ICE driver
- The C-SPY AVR Dragon driver
- The C-SPY AVR ONE! driver
- The C-SPY JTAGICE driver
- The C-SPY JTAGICE mkII driver
- The C-SPY JTAGICE3 driver

**Description** Use this option in the CLIB runtime environment to set a system breakpoint on the `putchar` label. This consumes a hardware breakpoint.

**See also** *Breakpoint consumers*, page 118



**Project>Options>Debugger>Driver>Driver 2>System breakpoints on>putchar**

## **--drv\_suppress\_download**

**Syntax** `--drv_suppress_download`

**For use with** All C-SPY hardware drivers.

**Description** Use this option to disable the downloading of code, preserving the current contents of the flash memory.



**Project>Options>Debugger>Driver>Driver 1>Suppress download**



**--drv\_use\_PDI**

Syntax	--drv_use_PDI
For use with	The C-SPY AVR Dragon driver The C-SPY JTAGICE mkII driver
Description	Use this option if you want the C-SPY driver to communicate with the device using the PDI interface.



**Project>Options>Debugger>Driver>Driver 1>Use PDI**

**--drv\_verify\_download**

Syntax	--drv_verify_download
For use with	All C-SPY hardware drivers.
Description	Use this option to verify that the downloaded code image can be read back from target memory with the correct contents.



**Project>Options>Debugger>Driver>Driver 1>Target Consistency Check>Verify All**

**--eeprom\_size**

Syntax	--eeprom_size= <i>size</i>
Parameters	<i>size</i> The size of the built-in EEPROM area in bytes.
For use with	All C-SPY drivers.
Description	Use this option to specify the size of the built-in EEPROM area. Do not use together with --cpu.



**Project>Options>General Options>Target>Utilize inbuilt EEPROM**

**--enhanced\_core**

Syntax	--enhanced_core
For use with	All C-SPY drivers.
Description	Use this option to enable the enhanced instruction set; the instructions MOVW, MUL, MULS, MULSU, FMUL, FMULS, FMULSU, LPM Rd, Z, LPM Rd, Z+, ELPM Rd, Z, ELPM Rd, Z+, and SPM.



**Project>Options>General Options>Target>Enhanced core**

**-f**

Syntax	-f <i>filename</i>
Parameters	<i>filename</i> A text file that contains the commands (default filename extension <code>.xcl</code> ).
For use with	<code>cspybat</code> This option can be placed both before and after the <code>--backend</code> option on the command line.
Description	Use this option to make <code>cspybat</code> read command line options from the named file.  In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character is treated like a space or tab character.  Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.



To set this option, use **Project>Options>Debugger>Extra Options**.

**--ice200\_restore\_EEPROM**

Syntax	--ice200_restore_EEPROM
For use with	The C-SPY ICE200 driver.

**Description** Use this option to restore the contents of the on-chip EEPROM data memory when the target board power is switched on again after having been switched off.



**Project>Options>Debugger>ICE200>ICE200>Restore EEPROM**

## --ice200\_single\_step\_timers

**Syntax** `--ice200_single_step_timers`

**For use with** The C-SPY ICE200 driver.

**Description** Use this option to allow the timers to single-step.



**Project>Options>Debugger>ICE200>ICE200>Single step timers**

## --jtagice\_clock

**Syntax** `--jtagice_clock=speed`

**Parameters**

*speed* The JTAG clock frequency in Hz. Possible values are 0-3570000 Hz.

**For use with** The C-SPY AVR Dragon driver  
The C-SPY JTAGICE driver  
The C-SPY JTAGICE mkII driver

**Description** Use this option to specify the speed of the JTAG clock.




**Project>Options>Debugger>Driver>Driver 1>JTAG Port>Frequency in Hz**


## --jtagice\_do\_hardware\_reset

**Syntax** `--jtagice_do_hardware_reset`


**For use with** The C-SPY Atmel-ICE driver  
The C-SPY AVR Dragon driver  
The C-SPY AVR ONE! driver

	The C-SPY JTAGICE driver
	The C-SPY JTAGICE mkII driver
	The C-SPY JTAGICE3 driver
Description	Use this option to make the hardware reset every time the debugger is reset.
	 <b>Project&gt;Options&gt;Debugger&gt;Driver&gt;Driver 2&gt;Hardware reset on C-SPY reset</b>


### --jtagice\_leave\_timers\_running

Syntax	--jtagice_leave_timers_running
For use with	The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE driver The C-SPY JTAGICE mkII driver
Description	Use this option to ensure that the timers always run, even if the application is stopped.
	 <b>Project&gt;Options&gt;Debugger&gt;Driver&gt;Driver 2&gt;Run timers in stopped mode</b>


### --jtagice\_preserve\_eeprom

Syntax	--jtagice_preserve_eeprom
For use with	The C-SPY Atmel-ICE driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option to preserve the EEPROM contents even if device is reprogrammed.
	 <b>Project&gt;Options&gt;Debugger&gt;Driver&gt;Driver 2&gt;Preserve EEPROM contents even if device is reprogrammed</b>

## --jtagice\_restore\_fuse

Syntax	--jtagice_restore_fuse
For use with	The C-SPY Atmel-ICE driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option to allow the debugger to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch decreases the life span of the chip.
	 <b>Project&gt;Options&gt;Debugger&gt;Driver&gt;Driver 2&gt;Restore fuses when ending debug session</b>

## --jtagicemkII\_use\_software\_breakpoints

Syntax	--jtagicemkII_use_software_breakpoints
For use with	The C-SPY Atmel-ICE driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option to make software breakpoints available.
	 <b>Project&gt;Options&gt;Debugger&gt;Driver&gt;Driver 2&gt;Enable software breakpoints</b>

## --leave\_running

Syntax	--leave_running
	Note that this option must be placed before the --backend option on the command line.

For use with	<code>cspybat</code>
Description	Makes <code>cspybat</code> start the execution on the target and then exits but leaves the target running.



To set a related option, choose:

**Project>Options>Debugger>Download>Attach to running target**

## --macro

Syntax	<code>--macro filename</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The C-SPY macro file to be used (filename extension <code>mac</code> ).
For use with	<code>cspybat</code>
Description	Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.
See also	<i>Briefly about using C-SPY macros</i> , page 208.



**Project>Options>Debugger>Setup>Setup macros>Use macro file**

## --macro\_param

Syntax	<code>--macro_param [param=value]</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>param = value</i> <i>param</i> is a parameter defined using the <code>__param</code> C-SPY macro construction. <i>value</i> is a value.
For use with	<code>cspybat</code>
Description	Use this option to assign a value to a C-SPY macro parameter. This option can be used more than once on the command line.
See also	<i>Macro parameters</i> , page 215.

**Project>Options>Debugger>Extra Options****-p**

Syntax	<code>-p filename</code>
Parameters	<i>filename</i> The device description file to be used.
For use with	All C-SPY drivers.
Description	Use this option to specify the device description file to be used.
See also	<i>Selecting a device description file</i> , page 52.

**Project>Options>Debugger>Setup>Device description file****--plugin**

Syntax	<code>--plugin filename</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The plugin file to be used (filename extension <code>dll</code> ).
For use with	<code>cspybat</code>
Description	Certain C/C++ standard library functions, for example <code>printf</code> , can be supported by C-SPY—for example, the C-SPY <b>Terminal I/O</b> window—instead of by real hardware devices. To enable such support in <code>cspybat</code> , a dedicated plugin module called <code>avrllibsupportbat.dll</code> located in the <code>avr\bin</code> directory must be used.  Use this option to include this plugin during the debug session. This option can be used more than once on the command line.  <b>Note:</b> You can use this option to include also other plugin modules, but in that case the module must be able to work with <code>cspybat</code> specifically. This means that the C-SPY plugin modules located in the <code>common\plugin</code> directory cannot normally be used with <code>cspybat</code> .

**Project>Options>Debugger>Plugins****--silent**

Syntax

`--silent`

Note that this option must be placed before the `--backend` option on the command line.

For use with

`cspybat`

Description

Use this option to omit the sign-on message.



This option is not available in the IDE.

**--timeout**

Syntax

`--timeout milliseconds`

Note that this option must be placed before the `--backend` option on the command line.

Parameters

*milliseconds*

The number of milliseconds before the execution stops.

For use with

`cspybat`

Description

Use this option to limit the maximum allowed execution time.



This option is not available in the IDE.

**-v**

Syntax

`-v {0|1|2|3|4|5|6}`

Parameters

- 0 A maximum of 256 bytes data and 8 Kbytes code. Default memory model: Tiny.
- 1 A maximum of 64 Kbytes data and 8 Kbytes code. Default memory model: Tiny.
- 2 A maximum of 256 bytes data and 128 Kbytes code. Default memory model: Tiny.
- 3 A maximum of 64 Kbytes data and 128 Kbytes code. Default memory model: Tiny.



	<ol style="list-style-type: none"><li>4 A maximum of 16 Mbytes data and 128 Kbytes code. Default memory model: Small.</li><li>5 A maximum of 64 Kbytes data and 8 Mbytes code. Default memory model: Small.</li><li>6 A maximum of 16 Mbytes data and 8 Mbytes code. Default memory model: Small.</li></ol>
For use with	All C-SPY drivers.
Description	Use this option to specify the processor configuration your application was compiled for. This option cannot be used together with <code>--cpu</code> .



**Project>Options>General Options>Target>Processor configuration**



# Part 4. Additional reference information

This part of the *C-SPY® Debugging Guide for AVR* includes these chapters:

- Debugger options
- Additional information on C-SPY drivers





# Debugger options

- Setting debugger options
- Reference information on debugger options
- Reference information on C-SPY hardware debugger driver options

---

## Setting debugger options

Before you start the C-SPY debugger you might need to set some options—both C-SPY generic options and options required for the target system (C-SPY driver-specific options).

### To set debugger options in the IDE:

- 1 Choose **Project>Options** to display the **Options** dialog box.
- 2 Select **Debugger** in the **Category** list.

For more information about the generic options, see *Reference information on debugger options*, page 294.

- 3 On the **Setup** page, make sure to select the appropriate C-SPY driver from the **Driver** drop-down list.
- 4 To set the driver-specific options, select the appropriate driver from the **Category** list. Depending on which C-SPY driver you are using, different options are available.

C-SPY driver	Available options pages
C-SPY AVR ONE! driver	AVR ONE! 1, page 298 AVR ONE! 2, page 301 Communication, page 302 Extra Options, page 303
C-SPY CCR driver	CCR, page 304 Serial Port, page 305 Extra Options, page 303
C-SPY ICE200 driver	ICE200, page 307 Serial Port, page 305 Extra Options, page 303

Table 35: Options specific to the C-SPY drivers you are using

<b>C-SPY driver</b>	<b>Available options pages</b>
C-SPY JTAGICE driver	<i>JTAGICE 1</i> , page 309 <i>JTAGICE 2</i> , page 311 <i>Serial Port</i> , page 305 <i>Extra Options</i> , page 303
C-SPY Atmel-ICE driver	<i>Atmel-ICE 1</i> , page 312 <i>Atmel-ICE 2</i> , page 314 <i>Communication</i> , page 302 <i>Extra Options</i> , page 303
C-SPY JTAGICE3 driver	<i>JTAGICE3 1</i> , page 316 <i>JTAGICE3 2</i> , page 318 <i>Communication</i> , page 302 <i>Extra Options</i> , page 303
C-SPY JTAGICE mk II driver	<i>JTAGICE mkII 1</i> , page 319 <i>JTAGICE mkII 2</i> , page 322 <i>Serial Port</i> , page 305 <i>Extra Options</i> , page 303
C-SPY Dragon driver	<i>Dragon 1</i> , page 323 <i>Dragon 2</i> , page 325 <i>Communication</i> , page 302 <i>Extra Options</i> , page 303
C-SPY Simulator	<i>Extra Options</i> , page 303
Third-party driver	<i>Third-Party Driver options</i> , page 326 <i>Extra Options</i> , page 303

*Table 35: Options specific to the C-SPY drivers you are using (Continued)*

- 5** To restore all settings to the default factory settings, click the **Factory Settings** button.
- 6** When you have set all the required options, click **OK** in the **Options** dialog box.

---

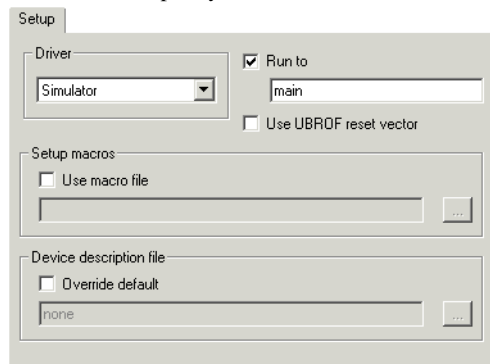
## Reference information on debugger options

Reference information about:

- Setup
- Images
- Plugins

## Setup

The **Setup** options select the C-SPY driver, the setup macro file, and device description file to use, and specify which default source code location to run to.



### Driver

Selects the C-SPY driver for the target system you have.

### Run to

Specifies the location C-SPY runs to when the debugger starts after a reset. By default, C-SPY runs to the `main` function.

To override the default location, specify the name of a different location you want C-SPY to run to. You can specify assembler labels or whatever can be evaluated as such, for example function names.

If the option is deselected, the program counter will contain the regular hardware reset address at each reset.

### Use UBROF reset vector

Makes the debugger use the reset vector specified as the entry label `__program_start`, see the *IDE Project Management and Building Guide*. By default, the reset vector is set to `0x0`.

### Setup macros

Registers the contents of a setup macro file in the C-SPY startup sequence. Select **Use macro file** and specify the path and name of the setup file, for example `SetupSimple.mac`. If no extension is specified, the extension `mac` is assumed. A browse button is available for your convenience.

## Device description file

A default device description file is selected automatically based on your project settings. To override the default file, select **Override default** and specify an alternative file. A browse button is available for your convenience.

For information about the device description file, see *Modifying a device description file*, page 56.

IAR-specific device description files for each AVR device are provided in the directory `avr\config` and have the filename extension `ddf`.

## Images

The **Images** options control the use of additional debug files to be downloaded.

The screenshot shows a dialog box titled "Images" with a light green background. It contains three vertically stacked sections. Each section starts with a checkbox labeled "Download extra image". Below each checkbox is a "Path:" text box followed by a small square browse button. Underneath the path box is an "Offset:" text box. To the right of the offset box is another checkbox labeled "Debug info only".

### Download extra Images

Controls the use of additional debug files to be downloaded:

#### Path

Specify the debug file to be downloaded. A browse button is available for your convenience.

#### Offset

Specify an integer that determines the destination address for the downloaded debug file.

#### Debug info only

Makes the debugger download only debug information, and not the complete debug file.

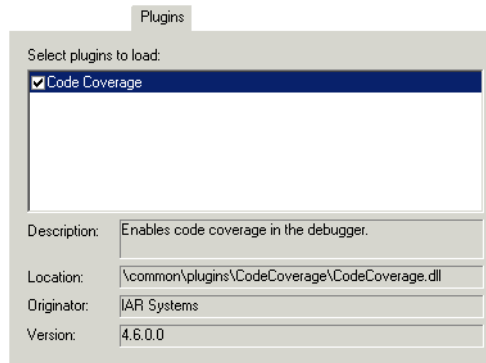
If you want to download more than three images, use the related C-SPY macro, see `__loadImage`, page 231.



For more information, see *Loading multiple images*, page 55.

## Plugins

The **Plugins** options select the C-SPY plugin modules to be loaded and made available during debug sessions.



### Select plugins to load

Selects the plugin modules to be loaded and made available during debug sessions. The list contains the plugin modules delivered with the product installation.

### Description

Describes the plugin module.

### Location

Informs about the location of the plugin module.

Generic plugin modules are stored in the `common\plugins` directory. Target-specific plugin modules are stored in the `avr\plugins` directory.

### Originator

Informs about the originator of the plugin module, which can be modules provided by IAR Systems or by third-party vendors.

### Version

Informs about the version number.



**Debug Port**

Selects the communication type. Choose between

<b>Auto detect</b>	Auto-detects JTAG or PDI. The JTAG Port options are not available in this mode. A JTAG device must be first in a JTAG chain.
<b>JTAG</b>	Specifies JTAG only mode.
<b>PDI</b>	Specifies PDI only mode.
<b>Frequency in Hz</b>	<p>Defines the debug port clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in kHz and can be any value from 1 to 65536. The default value is 100 kHz.</p> <p>A too small value (less than 28 kHz) might cause the communication to time out.</p> <p>A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.</p>

**JTAG Port**

Configures the JTAG port.

<b>Target device is part of a JTAG daisy chain</b>	If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.
<b>Devices Before</b>	Specify the number of JTAG data bits before the device in the JTAG chain.
<b>Devices After</b>	Specify the number of JTAG data bits after the device in the JTAG chain.
<b>Instruction bits Before</b>	Specify the number of JTAG instruction register bits before the device in the JTAG chain.
<b>Instruction bits After</b>	Specify the number of JTAG instruction register bits after the device in the JTAG chain.

## Download control

Controls the download.

### Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify all** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

### Allow download to RAM

Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.

### Target consistency check

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:

**None**, target consistency check is not performed.

**Verify Boundaries**, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.

**Verify All**, checks every byte after loading. This is a slow, but complete, check of the memory.

## AVR ONE! 2

The **AVR ONE! 2** options control the C-SPY driver for AVR ONE!.

AVR ONE! 2

Run timers in stopped mode

Preserve EEPROM contents even if device is reprogrammed

Hardware reset on C-SPY reset

Restore fuses when ending debug session

Enable software breakpoints

System breakpoints on

exit

putchar

getchar

Preserve FLASH

None

Boot Area

Application Area

### Run timers in stopped mode

Runs the timers even if the program is stopped.

### Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

### Hardware reset on C-SPY reset

Causes a reset in the debugger to also do a hardware reset by pulling down the nSRST signal.

### Restore fuses when ending debug session

Enables C-SPY to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch will decrease the life span of the chip.

### Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 113.

### System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options `exit`, `putchar`, and `getchar`, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

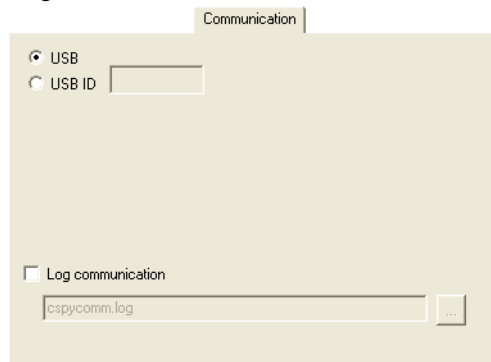
For more information, see *Breakpoint consumers*, page 118.

### Preserve FLASH

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

## Communication

The **Communication** options control the C-SPY driver for AVR ONE!, JTAGICE3, or Dragon.



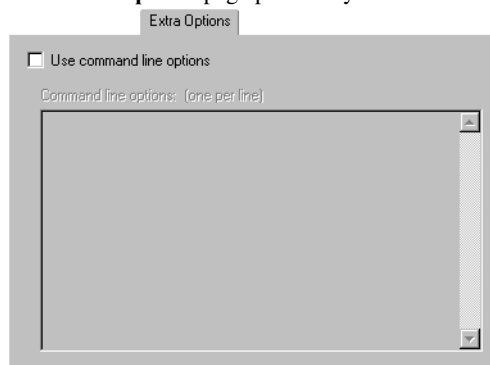
These options are available:

- |               |  |
|---------------|--|
| <b>USB</b>    | Specifies single emulator mode. Use this option for the USB port and if you have one device connected to your host computer.   |
| <b>USB ID</b> | Specifies multi-emulator mode. Use this option for the USB port and if you have more than one device connected to your host computer. Specify the serial number of the device that you want to connect to, or the USB ID visible in the Log Messages window. The serial number, for example <b>ONE00737</b> , is printed on the tag underneath the device. |

**Log communication** Logs the communication between C-SPY and the target system to the specified log file, which can be useful for troubleshooting purposes. The communication will be logged in the file `cspycomm.log` located in the current working directory. If required, use the browse button to choose a different file.

## Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.



### Use command line options

Specify command line arguments that are not supported by the IDE to be passed to C-SPY.

Note that it is possible to use the `/args` option to pass command line arguments to the debugged application.

Syntax: `/args arg0 arg1 ...`

Multiple lines with `/args` are allowed, for example:

```
/args --logfile log.txt
```

```
/args --verbose
```

If you use `/args`, these variables must be defined in your application:

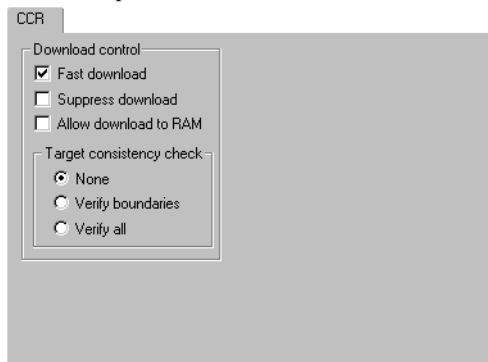
```
/* __argc, the number of arguments in __argv. */
__no_init int __argc;

/* __argv, an array of pointers to strings that holds the
arguments; must be large enough to fit the number of
parameters.*/
__no_init const char * __argv[MAX_ARGS];

/* __argvbuf, a storage area for __argv; must be large enough to
hold all command line parameters. */
__no_init __root char __argvbuf[MAX_ARG_SIZE];
```

## CCR

The CCR options control the C-SPY driver for CCR.



### Download control

Controls the download.

#### Fast download

Enables fast downloading of your application. By default, this option is enabled. If this option is disabled, downloading will take more time because a more robust protocol with error-checking is used. However, this should only be necessary if the ROM-monitor is not fast enough to process the data stream or, for example, if the communication cable is insufficiently shielded. For more information, see *Optimizing downloads*, page 343.



<b>Suppress download</b>	<p>Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.</p> <p>If this option is combined with the <b>Verify all</b> option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.</p>
<b>Allow download to RAM</b>	<p>Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.</p>
<b>Target consistency check</b>	<p>Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:</p> <p><b>None</b>, target consistency check is not performed.</p> <p><b>Verify Boundaries</b>, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.</p> <p><b>Verify All</b>, checks every byte after loading. This is a slow, but complete, check of the memory.</p>

## Serial Port

The Serial Port options determine how the serial port should be used.

Serial Port

Default communication

COM 1 Port COM 1

Baud 9600

Parity None

Data bits 8 data bits

Stop bits 1 stop bit

Handshaking None

Log communication

cspycomm.log

**Note:** This dialog box is disabled if you select any of the options **Default communication** on the **JTAGICE 1** page, or the **High speed** option on the **ICE200** page.

**Default communication**

Sets the default communication to use the port you specify and use it at 38400 baud.

**Note:** This option is only available for JTAGICE mkII.

**Port**

Selects one of the supported ports: **COM1** (default), **COM2**, ..., **COM32**.

**Baud**

Selects one of these speeds: **9600** (default for CCR), **14400**, **19200** (default for ICE200), **38400** (default for JTAGICE), **57600**, **115200** (default for JTAGICE mkII).

For the ICE emulators, C-SPY always tries to connect with the default baud when making the first contact with the target system. When contact has been established, the selected rate will be used.

If the debug session is terminated unexpectedly (by a fatal error, for instance), you might have to switch the emulator on and off to make it reconnect—first at default rate and then at the selected rate.

For the CCR, C-SPY tries to connect with the selected baud rate when making the first contact with the ROM-monitor.

**Parity**

Selects the parity; only **None** is allowed.

**Data bits**

Selects the number of data bits; only **8 data bits** is allowed.

**Stop bits**

Selects the number of stop bits: **1 stop bit** or **2 stop bits**.

**Handshaking**

Selects the handshaking method, **None** or **RTSCTS**.

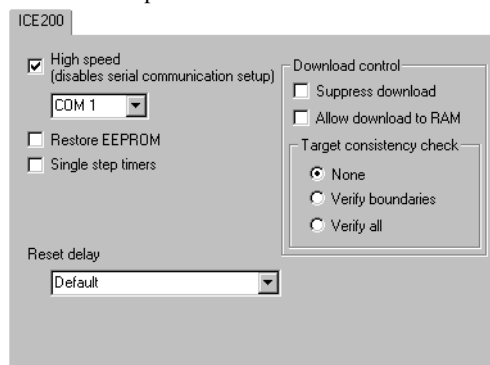
**Log communication**

Logs the communication between C-SPY and the target system to the specified log file, which can be useful for troubleshooting purposes. The communication will be logged in

the file `cspycomm.log` located in the current working directory. If required, use the browse button to locate a different file.

## ICE200

The **ICE200** options control the C-SPY driver for ICE200.



Note that you can make temporary changes to the ICE200 options during debug sessions; see *ICE200 Options dialog box*, page 340.

### High speed

Specifies the communication port. The ports supported are COM1, COM2, COM3, and COM4. COM1 is the default port.

It is recommended that you use the **High speed** option to specify the serial communication setup.

If this check box is selected, the communications options on the **Serial Port** page are disabled, and the default high speed communication is used. To enable the options on the **Serial Port** page, deselect this check box.

### Restore EEPROM

Some AVR devices have on-chip EEPROM data memory. The ICE200 emulates the EEPROM by using an SRAM replacement inside the AVR emulator chip. This is done to eliminate problems with EEPROM write endurance. However, by doing so, a new problem is introduced because a power loss on the target board will result in loss of the data stored in the SRAM that emulates the EEPROM.

A two-step solution handles the power loss situation. First select the **Restore EEPROM** option. Then, before removing the power, take a snapshot of the EEPROM contents by choosing **ICE200>EEPROM snapshot**. This will tell the ICE200 main board to read all EEPROM data into a non-volatile buffer. When target board power is switched off

and then on again, the ICE200 will restore the contents of the buffer to the SRAM before starting code execution.

### Single step timers

Enables single stepping of the timers. If deselected, the timers will continue to count (if internally enabled) even after the program execution has stopped. All other peripherals (SPI/UART/EEPROM/PORTs) will continue to operate when the program execution is stopped.

This feature allows cycle-by-cycle debugging of the timer counter value, which is useful for event timing. However, in many cases, stopping the counter operation while debugging is undesirable. One example is when the timer is used in PWM mode. Stopping the timer in this case might damage the equipment that is being controlled by the PWM output.

Note that timer interrupts (if any) will not be handled before execution is resumed.

### Download control

Controls the download.

#### Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify all** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

#### Allow download to RAM

Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.

### Target consistency check

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:

**None**, target consistency check is not performed.

**Verify Boundaries**, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.

**Verify All**, checks every byte after loading. This is a slow, but complete, check of the memory.

## JTAGICE I

The **JTAGICE 1** options control the C-SPY JTAGICE driver.

### Default communication

Sets the default communication to use the COM1 port and use it at 38400 baud.

If this check box is selected, the communications options on the **Serial port** page are disabled. To enable the options on the **Serial port** page, deselect this check box.

## JTAG Port

Configures the JTAG port.

<b>Frequency in Hz</b>	<p>Defines the JTAG clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in Hz and can be any value from 5000 to 1000000. The frequency value is rounded down to nearest available in the JTAGICE. The default value is 100 kHz.</p> <p>A too small value (less than 28000) might cause the communication to time out. The value must not be greater than 1/4 of the target CPU clock frequency.</p> <p>A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.</p>
<b>Target device is part of a JTAG daisy chain</b>	<p>If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.</p>
<b>Devices Before</b>	<p>Specify the number of JTAG data bits before the device in the JTAG chain.</p>
<b>Devices After</b>	<p>Specify the number of JTAG data bits after the device in the JTAG chain.</p>
<b>Instruction bits Before</b>	<p>Specify the number of JTAG instruction register bits before the device in the JTAG chain.</p>
<b>Instruction bits After</b>	<p>Specify the number of JTAG instruction register bits after the device in the JTAG chain.</p>

## Download control

Controls the download.

<b>Suppress download</b>	<p>Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.</p> <p>If this option is combined with the <b>Verify all</b> option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.</p>
--------------------------	--

**Allow download to RAM**

Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.

**Target consistency check**

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:

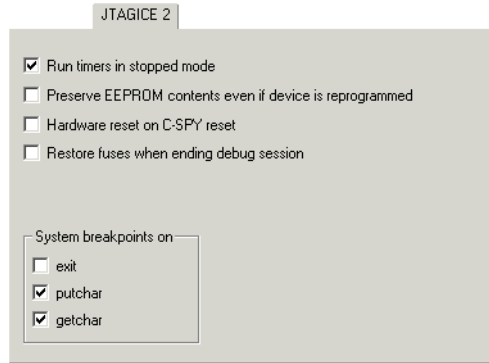
**None**, target consistency check is not performed.

**Verify Boundaries**, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.

**Verify All**, checks every byte after loading. This is a slow, but complete, check of the memory.

## JTAGICE 2

The **JTAGICE 2** options control the JTAGICE driver.

**Run timers in stopped mode**

Runs the timers even if the program is stopped.

**Preserve EEPROM contents even if device is reprogrammed**

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

**Hardware reset on C-SPY reset**

Causes a reset in the debugger to also do a hardware reset by pulling down the nSRST signal.

**Restore fuses when ending debug session**

Enables C-SPY to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch will decrease the life span of the chip.

**System breakpoints on**

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options `exit`, `putchar`, and `getchar`, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

For more information, see *Breakpoint consumers*, page 118.

## Atmel-ICE 1

The **Atmel-ICE 1** options control the C-SPY driver for Atmel-ICE.

**Debug Port**

Selects the communication type. Choose between:

**Auto detect**

Auto-detects JTAG or PDI. The JTAG Port options are not available with this setting. A JTAG device must be first in a JTAG chain.



<b>JTAG</b>	Specifies JTAG only mode.
<b>PDI</b>	Specifies PDI only mode.
<b>Frequency in Hz</b>	<p>Defines the debug port clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in kHz and can be any value from 1 to 65536. The default value is 100 kHz.</p> <p>A too small value (less than 28 kHz) might cause the communication to time out.</p> <p>A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.</p>

### JTAG Port

Configures the JTAG port.

<b>Target device is part of a JTAG daisy chain</b>	If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.
<b>Devices Before</b>	Specify the number of JTAG data bits before the device in the JTAG chain.
<b>Devices After</b>	Specify the number of JTAG data bits after the device in the JTAG chain.
<b>Instruction bits Before</b>	Specify the number of JTAG instruction register bits before the device in the JTAG chain.
<b>Instruction bits After</b>	Specify the number of JTAG instruction register bits after the device in the JTAG chain.

### Download control

Controls the download.

<b>Suppress download</b>	<p>Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.</p> <p>If this option is combined with the <b>Verify all</b> option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.</p>
--------------------------	--

**Allow download to RAM**

Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.

**Target consistency check**

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:

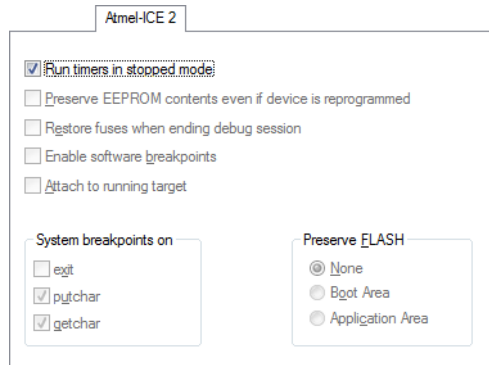
**None**, target consistency check is not performed.

**Verify Boundaries**, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.

**Verify All**, checks every byte after loading. This is a slow, but complete, check of the memory.

## Atmel-ICE 2

The **Atmel-ICE 2** options control the C-SPY driver for Atmel-ICE.



**Run timers in stopped mode**

Runs the timers even if the program is stopped.

**Preserve EEPROM contents even if device is reprogrammed**

Erases flash memory before download. If this option is deselected, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

**Restore fuses when ending debug session**

Enables C-SPY to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch will decrease the life span of the chip.

**Enable software breakpoints**

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 113.

**Attach to running target**

Makes the debugger attach to a running application at its current location, without resetting or halting the target system. To avoid unexpected behavior when using this option, the **Debugger>Setup** option **Run to** should be deselected.

**System breakpoints on**

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY **Terminal I/O** window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options **exit**, **putchar**, and **getchar**, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

For more information, see *Breakpoint consumers*, page 118.

**Preserve FLASH**

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

## JTAGICE3 I

The **JTAGICE3 1** options control the C-SPY driver for JTAGICE3.

### Debug Port

Selects the communication type. Choose between

**Auto detect** Auto-detects JTAG or PDI. The JTAG Port options are not available with this setting. A JTAG device must be first in a JTAG chain.

**JTAG** Specifies JTAG only mode.

**PDI** Specifies PDI only mode.

**Frequency in Hz** Defines the debug port clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in kHz and can be any value from 1 to 65536. The default value is 100 kHz.

A too small value (less than 28 kHz) might cause the communication to time out.

A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.

### JTAG Port

Configures the JTAG port.

**Target device is part of a JTAG daisy chain** If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.

<b>Devices Before</b>	Specify the number of JTAG data bits before the device in the JTAG chain.
<b>Devices After</b>	Specify the number of JTAG data bits after the device in the JTAG chain.
<b>Instruction bits Before</b>	Specify the number of JTAG instruction register bits before the device in the JTAG chain.
<b>Instruction bits After</b>	Specify the number of JTAG instruction register bits after the device in the JTAG chain.

### Download control

Controls the download.

<b>Suppress download</b>	<p>Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.</p> <p>If this option is combined with the <b>Verify all</b> option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.</p>
<b>Allow download to RAM</b>	Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.
<b>Target consistency check</b>	<p>Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:</p> <p><b>None</b>, target consistency check is not performed.</p> <p><b>Verify Boundaries</b>, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.</p> <p><b>Verify All</b>, checks every byte after loading. This is a slow, but complete, check of the memory.</p>

## JTAGICE3 2

The **JTAGICE3 2** options control the C-SPY driver for JTAGICE3.

### Run timers in stopped mode

Runs the timers even if the program is stopped.

### Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

### Restore fuses when ending debug session

Enables C-SPY to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch will decrease the life span of the chip.

### Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 113.

### Attach to running target

Makes the debugger attach to a running application at its current location, without resetting or halting the target system. To avoid unexpected behavior when using this option, the **Debugger>Setup** option **Run to** should be deselected.

### System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options **exit**, **putchar**, and **getchar**, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

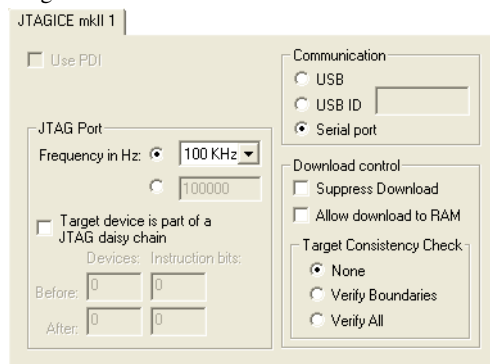
For more information, see *Breakpoint consumers*, page 118.

### Preserve FLASH

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

## JTAGICE mkII I

The **JTAGICE mkII 1** options control the C-SPY drivers for JTAGICE mkII and Dragon.



### Use PDI

Enables communication with the device using the PDI interface.

### Communication

Selects the communication type. Choose between:

#### USB

Selects the USB port. Use this setting if you have one AVR JTAGICE mkII device connected to your host computer.

#### USB ID

Selects the USB port. Use this setting if you have more than one AVR JTAGICE mkII device connected to your host computer. Specify the serial number of the device that you want to connect to. The serial number is printed on the tag underneath the device.

**Serial port** Selects the serial port. To configure the serial port, select the **Serial Port** page in the **Options** dialog box and then choose a port from the **Default communication** drop-down list. By default, the COM1 port is used at 38400 baud.

## JTAG Port

Configures the JTAG port.

**Frequency in Hz** Defines the JTAG clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in Hz and can be any value from 5000 to 1000000. The frequency value is rounded down to nearest available in the JTAGICE. The default value is 100 kHz.

A too small value (less than 28000) might cause the communication to time out. The value must not be greater than 1/4 of the target CPU clock frequency.

A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.

**Target device is part of a JTAG daisy chain** If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.

**Devices Before** Specify the number of JTAG data bits before the device in the JTAG chain.

**Devices After** Specify the number of JTAG data bits after the device in the JTAG chain.

**Instruction bits Before** Specify the number of JTAG instruction register bits before the device in the JTAG chain.

**Instruction bits After** Specify the number of JTAG instruction register bits after the device in the JTAG chain.



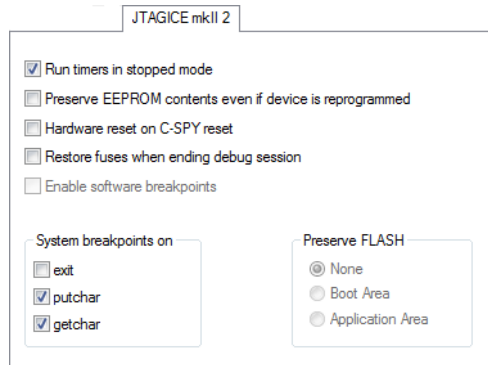
## Download control

Controls the download.

- Suppress download** Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.
- If this option is combined with the **Verify all** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.
- Allow download to RAM** Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.
- Target consistency check** Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:
- None**, target consistency check is not performed.
  - Verify Boundaries**, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.
  - Verify All**, checks every byte after loading. This is a slow, but complete, check of the memory.

## JTAGICE mkII 2

The **JTAGICE mkII 2** options control the C-SPY drivers for JTAGICE mkII and Dragon.



JTAGICE mkII 2

Run timers in stopped mode

Preserve EEPROM contents even if device is reprogrammed

Hardware reset on C-SPY reset

Restore fuses when ending debug session

Enable software breakpoints

System breakpoints on

exit

putchar

getchar

Preserve FLASH

None

Boot Area

Application Area

### Run timers in stopped mode

Runs the timers even if the program is stopped.

### Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

### Hardware reset on C-SPY reset

Causes a reset in the debugger to also do a hardware reset by pulling down the nSRST signal.

### Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 113.

### System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options `exit`, `putchar`, and `getchar`, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

For more information, see *Breakpoint consumers*, page 118.

## Preserve FLASH

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

## Dragon 1

The **Dragon 1** options control the C-SPY drivers for Dragon.

## Use PDI

Enables communication with the device using the PDI interface.

## JTAG Port

Configures the JTAG port.

### Frequency in Hz

Defines the JTAG clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in Hz and can be any value from 5000 to 1000000. The frequency value is rounded down to nearest available in the JTAGICE. The default value is 100 kHz.

A too small value (less than 28000) might cause the communication to time out. The value must not be greater than 1/4 of the target CPU clock frequency.

A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.

### Target device is part of a JTAG daisy chain

If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.

<b>Devices Before</b>	Specify the number of JTAG data bits before the device in the JTAG chain.
<b>Devices After</b>	Specify the number of JTAG data bits after the device in the JTAG chain.
<b>Instruction bits Before</b>	Specify the number of JTAG instruction register bits before the device in the JTAG chain.
<b>Instruction bits After</b>	Specify the number of JTAG instruction register bits after the device in the JTAG chain.

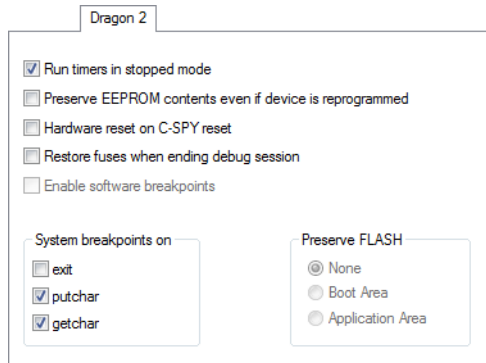
### Download control

Controls the download.

<b>Suppress download</b>	<p>Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.</p> <p>If this option is combined with the <b>Verify all</b> option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.</p>
<b>Allow download to RAM</b>	Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.
<b>Target consistency check</b>	<p>Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:</p> <p><b>None</b>, target consistency check is not performed.</p> <p><b>Verify Boundaries</b>, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.</p> <p><b>Verify All</b>, checks every byte after loading. This is a slow, but complete, check of the memory.</p>

## Dragon 2

The **Dragon 2** options control the C-SPY drivers for Dragon.



Dragon 2

Run timers in stopped mode

Preserve EEPROM contents even if device is reprogrammed

Hardware reset on C-SPY reset

Restore fuses when ending debug session

Enable software breakpoints

System breakpoints on

exit

putchar

getchar

Preserve FLASH

None

Boot Area

Application Area

### Run timers in stopped mode

Runs the timers even if the program is stopped.

### Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

### Hardware reset on C-SPY reset

Causes a reset in the debugger to also do a hardware reset by pulling down the nSRST signal.

### Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 113.

### System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

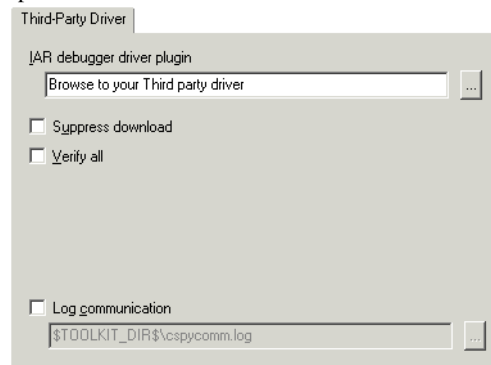
Select or deselect the options `exit`, `putchar`, and `getchar`, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

For more information, see *Breakpoint consumers*, page 118.

## Third-Party Driver options

The **Third-Party Driver** options are used for loading any driver plugin provided by a third-party vendor. These drivers must be compatible with the C-SPY debugger driver specification.



### IAR debugger driver plugin

Specify the file path to the third-party driver plugin DLL file. A browse button is available for your convenience.

### Suppress download

Disables the downloading of code, while preserving the present content of the flash. This command is useful if you need to exit C-SPY for a while and then continue the debug session without downloading code. The implicit RESET performed by C-SPY at startup is not disabled though.

If this option is combined with **Verify all**, the debugger will read your application back from the flash memory and verify that it is identical with the application currently being debugged.

This option can be used if it is supported by the third-party driver.

### Verify all

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Every byte is checked after it is loaded. This is a slow but complete check of the memory. This option can be used if is supported by the third-party driver.

**Log communication**

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required. This option can be used if is supported by the third-party driver.





# Additional information on C-SPY drivers

This chapter describes the additional menus and features provided by the C-SPY® drivers. You will also find some useful hints about resolving problems.

---

## Reference information on C-SPY driver menus

Reference information about:

- *C-SPY driver*, page 329
- *Simulator menu*, page 330
- *JTAGICE menu*, page 331
- *JTAGICE mkII menu*, page 331
- *Dragon menu*, page 332
- *Atmel-ICE menu*, page 332
- *JTAGICE3 menu*, page 333
- *AVR ONE! menu*, page 333
- *ICE200 menu*, page 334
- *CCR menu*, page 334

### **C-SPY driver**

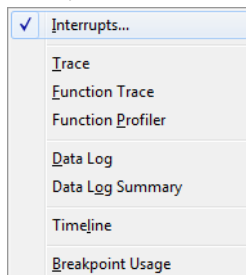
Before you start the C-SPY debugger, you must first specify a C-SPY driver in the **Options** dialog box, using the option **Debugger>Setup>Driver**.

When you start a debug session, a menu specific to that C-SPY driver will appear on the menu bar, with commands specific to the driver.

When we in this guide write “choose *C-SPY driver*>” followed by a menu command, *C-SPY driver* refers to the menu. If the feature is supported by the driver, the command will be on the menu.

## Simulator menu

When you use the simulator driver, the **Simulator** menu is added to the menu bar.



### Menu commands

These commands are available on the menu:

#### Interrupts

Displays a dialog box where you can manage interrupts, see *Interrupts dialog box*, page 204.

#### Trace

Opens a window which displays the collected trace data, see *Trace window*, page 168.

#### Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 170.

#### Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 188.

#### Data Log

Opens a window which logs accesses to up to four different memory locations or areas, see *Data Log window*, page 107.

#### Data Log Summary

Opens a window which displays a summary of data accesses to specific memory location or areas, see *Data Log Summary window*, page 109.

#### Timeline

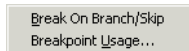
Opens a window which gives a graphical view of various kinds of information on a timeline, see *Timeline window*, page 171.

**Breakpoint Usage**

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

**JTAGICE menu**

When you are using the C-SPY JTAGICE driver, the **JTAGICE** menu is added to the menu bar.



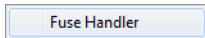
These commands are available on the menu:

**Break On Branch/Skip** Stops execution just after each branch instruction.

**Breakpoint Usage** Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

**JTAGICE mkII menu**

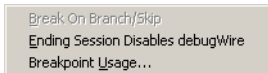
When you are using the C-SPY JTAGICE mkII driver, the **JTAGICE mkII** menu is added to the menu bar. Before the debugger is started, the menu looks like this:



This command is available on the menu:

**Fuse Handler** Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 335.

When the debugger is running, the menu looks like this:



These commands are available on the menu:

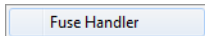
**Break On Branch/Skip** Stops execution just after each branch instruction.

**Ending Session Disables debugWire** Disables the use of debugWIRE before ending the debug session.

**Breakpoint Usage** Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

## Dragon menu

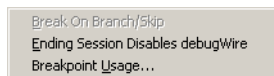
When you are using the C-SPY Dragon driver, the **Dragon** menu is added to the menu bar. Before the debugger is started, the menu looks like this:



This command is available on the menu:

**Fuse Handler** Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 335.

When the debugger is running, the menu looks like this:



These commands are available on the menu:

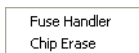
**Break On Branch/Skip** Stops execution just after each branch instruction.

**Ending Session Disables debugWire** Disables the use of debugWIRE before ending the debug session.

**Breakpoint Usage** Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

## Atmel-ICE menu

When you are using the C-SPY Atmel-ICE driver, the **Atmel-ICE** menu is added to the menu bar. Before the debugger is started, the menu looks like this:

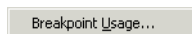


These commands are available on the menu:

**Fuse Handler** Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 338.

**Chip Erase** Performs a chip erase, that is, erases flash memory, EEPROM, and lock bits. For more information, see the documentation for the target you are using.

When the debugger is running, the menu looks like this:

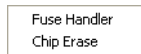


This command is available on the menu:

**Breakpoint Usage** Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

## JTAGICE3 menu

When you are using the C-SPY JTAGICE3 driver, the **JTAGICE3** menu is added to the menu bar. Before the debugger is started, the menu looks like this:

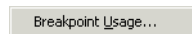


These commands are available on the menu:

**Fuse Handler** Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 338.

**Chip Erase** Performs a chip erase, that is, erases flash memory, EEPROM, and lock bits. For more information, see the documentation for the target you are using.

When the debugger is running, the menu looks like this:

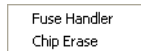


This command is available on the menu:

**Breakpoint Usage** Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

## AVR ONE! menu

When you are using the C-SPY AVR ONE! driver, the **AVR ONE!** menu is added to the menu bar. Before the debugger is started, the menu looks like this:

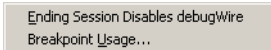


These commands are available on the menu:

**Fuse Handler** Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 338.

**Chip Erase** Performs a chip erase, that is, erases flash memory, EEPROM, and lock bits. For more information, see the documentation for the target you are using.

When the debugger is running, the menu looks like this:



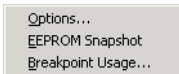
This command is available on the menu:

**Ending Session Disables debugWire** Disables the use of debugWIRE before ending the debug session.

**Breakpoint Usage** Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

## ICE200 menu

When you are using the C-SPY ICE200 driver, the **ICE200** menu is added to the menu bar.



These commands are available on the menu:

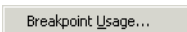
**Options** Displays a dialog box to temporarily change the settings of the ICE200 options during the debug session; see *ICE200 Options dialog box*, page 340.

**EEPROM Snapshot** Saves the contents of the EEPROM into a buffer. The saved contents will be restored to the EEPROM (emulated by SRAM in ICE200) at the next ICE200 power-up. See also *Restore EEPROM*, page 307.

**Breakpoint Usage** Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

## CCR menu

When you are using the C-SPY CCR driver, the **CCR** menu is added to the menu bar.



This command is available on the menu:

**Breakpoint Usage** Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 126.

## Reference information on the C-SPY hardware debugger drivers

This section gives additional reference information on the C-SPY hardware debugger drivers, reference information not provided elsewhere in this documentation.

Reference information about:

- *Fuse Handler dialog box*, page 335 (JTAGICE mkII and Dragon)
- *Fuse Handler dialog box*, page 338 (Atmel-ICE, AVR ONE!, and JTAGICE3)
- *ICE200 Options dialog box*, page 340

### Fuse Handler dialog box

The **Fuse Handler** dialog box is available from the **JTAGICE mkII** menu or the **Dragon** menu, respectively.

**Fuse Handler**

Overview

	Lock Bits	Low Fuse	High Fuse	Extended Fuse
New Value:	0xFF	0xE2	0x9F	0xF9
Old Value:	0xFF	0xA2	0x9F	0xF9

Device ID: 0x9330A

Buttons: Read Fuses, Program Fuses, Close

Lock Bits | Low Fuse | High Fuse | Extended Fuse

Old Value: 0xE2 | New Value: 0xA2

Divide clock by 8 internally; [CKDIV8=0]

Clock output on PORTB0; [CKOUT=0]

Int. RC Osc. 8 MHz; Start-up time PWRDWN/RESET: 6 CK/14 CK + 65 ms; [CKSEL=0010 SUT=10]; default value

Log Messages

Temporarily disabling the DWEN fuse. To re-enable it, switch off the target board power and then switch it on again.

AVR JTAGICE mkII, H/W version: 0x0000, S/W version: 0x0409 0x0418, Device id: 0x9330A  
Using debugWire, Target voltage: 5.118 V, CPU: ATmega88  
Starting to read fuses...  
Succeeded to read fuses

**Note:** To use the fuse handler, the JTAG Enable fuse must be enabled on one of the pages. If a debugWIRE interface is used, it will be temporarily disabled while using the fuse handler. Before you start debugging and after programming the fuses, you must enable the interface again. The JTAGICE mkII/Dragon debugger driver will guide you through this.

The fuse handler provides programming possibilities of the device-specific on-chip fuses and lock bits via JTAGICE mkII/Dragon.

Because different devices have different features, the available options and possible settings depend on which device is selected. However, the available options are divided into a maximum of four groups: **Lock Bits**, **Low Fuse**, **High Fuse**, and **Extended Fuse**. For detailed information about the fuse settings, see the device-specific documentation provided by Atmel® Corporation.

When you open the **Fuse Handler** dialog box, the device-specific fuses are read and the dialog box will reflect the fuse values.

## Requirements

One of these alternatives:

- The C-SPY JTAGICE mkII driver
- The C-SPY Dragon driver.

## Overview

Displays an overview of the fuse settings for each fuse group.

<b>New Value</b>	Displays the value of the fuses reflecting the user-defined settings. In other words, the value of the fuses after they have been programmed.
<b>Old Value</b>	Displays the last read value of the fuses.

## Lock Bits, Low Fuse, High Fuse, Extended Fuse pages

Contain the available options for each group of fuses. The options and possible settings depend on the selected device.

<b>Old Value</b>	Displays the last read value of the on-chip fuses on the device.
<b>New Value</b>	Displays the value of the fuses reflecting the user-defined settings. This text field is editable.



To specify the fuse settings, you can either use the **New Value** text field or use the options.

Selecting an option means that this fuse should be enabled/programmed, which means that the on-chip fuse is set to zero.

### **Device ID**

Displays the device ID that has been read from the device.

### **Read Fuses**

Reads the on-chip fuses from the device and the **Before** text fields will be updated accordingly.

### **Program Fuses**

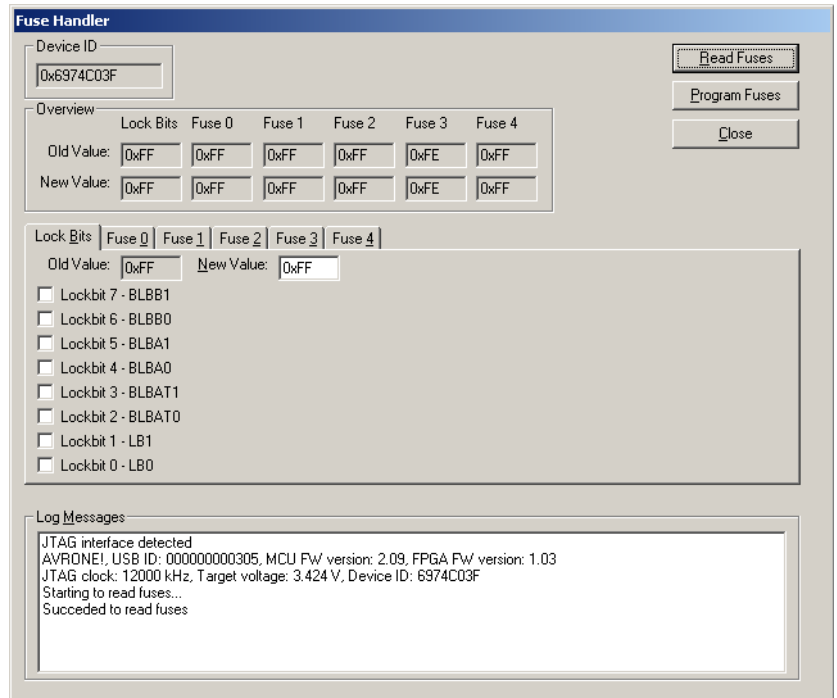
Programs the new fuse values—displayed in the **After** text box—to the on-chip fuses.

### **Log Messages**

Displays the device information, and status and diagnostic messages for all read/program operations.

## Fuse Handler dialog box

The **Fuse Handler** dialog box is available from the **Atmel-ICE** menu, the **AVR ONE!** menu, or the **JTAGICE3** menu, respectively.



**Note:** To use the fuse handler, the JTAG Enable fuse must be enabled on one of the tabs or you can use PDI, see *Atmel-ICE 1*, page 312, *AVR ONE! 1*, page 298, or *JTAGICE3 1*, page 316, specifically the information about the **Debug Port** option. The debugger driver will guide you through this.

The fuse handler provides programming possibilities of the device-specific on-chip fuses and lock bits via the debug probe.

Because different devices have different features, the available options and possible settings depend on which device is selected. However, the available options are divided into a maximum of six groups: **Lock Bits**, **Fuse 0**, **Fuse 1**, **Fuse 2**, **Fuse 3**, and **Fuse 4**. For detailed information about the fuse settings, see the device-specific documentation provided by Atmel® Corporation.

When you open the **Fuse Handler** dialog box, the device-specific fuses are read and the dialog box will reflect the fuse values.

**Requirements**

One of these alternatives:

- The C-SPY Atmel-ICE driver
- The C-SPY AVR ONE! driver
- The C-SPY JTAGICE3 driver.

**Device ID**

Displays the device ID that has been read from the device.

**Overview**

Displays an overview of the fuse settings for each fuse group.

<b>New Value</b>	Displays the value of the fuses reflecting the user-defined settings. In other words, the value of the fuses after they have been programmed.
<b>Old Value</b>	Displays the last read value of the fuses.

**Lock Bits, Fuse 0, Fuse 1, Fuse 2, Fuse 3, and Fuse 4 pages**

Contain the available options for each group of fuses. The options and possible settings depend on the selected device.

<b>Old Value</b>	Displays the last read value of the on-chip fuses on the device.
<b>New Value</b>	Displays the value of the fuses reflecting the user-defined settings. This text field is editable.

To specify the fuse settings, you can either use the **New Value** text field or use the options.

Selecting an option means that this fuse should be enabled/programmed, which means that the on-chip fuse is set to zero.

**Read Fuses**

Reads the on-chip fuses from the device and the **Before** text fields will be updated accordingly.

**Program Fuses**

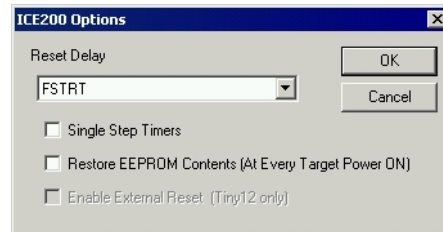
Programs the new fuse values—displayed in the **After** text box—to the on-chip fuses.

## Log Messages

Displays the device information, and status and diagnostic messages for all read/program operations.

## ICE200 Options dialog box

The **ICE200 Options** dialog box is available from the **ICE200** menu.



Use this dialog box to make temporary changes to the ICE200 options during debug sessions.

## Requirements

The C-SPY ICE200 driver.

## Reset Delay

Sets up a reset delay time. Select a clock from the drop-down list.

Many AVR microcontrollers have fuse bits for setting the reset delay time. The reset delay is necessary for the clock oscillator to stabilize. The time it takes for the oscillator to stabilize depends on the crystal or the resonator. If an external clock source is used, the reset delay can be set to only a few clock cycles. The reset delay fuse bits (CKSEL/FSTRT depending on the device) can be set or cleared by a parallel or serial programmer in an actual device. In the C-SPY ICE200, they are controlled by this option.

For more information about the reset delay fuses, refer to the data sheets for ICE200.

Available items for AT90S8515, AT90S8535 and AT90S2323:

- FSTRT (Fast Start)
- Normal

Available items for AT90S4433/2333:

- Ceramic resonator
- Ceramic resonator, BOD enabled

- Ceramic resonator, fast rising power
- Crystal oscillator
- Crystal oscillator, BOD enabled
- Crystal oscillator, fast rising power
- External clock, BOD enabled
- External clock, slowly rising power

Available items for ATtiny12:

- 0 External clock
- 1 External clock
- (2)Internal RC oscillator
- (3)Internal RC oscillator
- (4)Internal RC oscillator
- 5 External RC oscillator
- 6 External RC oscillator
- 7 External RC oscillator
- 8 External low-frequency crystal
- 9 External low-frequency crystal
- 10 External clock/Ceramic resonator
- 11 External clock/Ceramic resonator
- 12 External clock/Ceramic resonator
- 13 External clock/Ceramic resonator
- 14 External clock/Ceramic resonator
- 15 External clock/Ceramic resonator

### Single Step Timers

Enables single stepping of the timers. If deselected, the timers will continue to count (if internally enabled) even after the program execution has stopped. All other peripherals (SPI/UART/EEPROM/PORTs) will continue to operate when the program execution is stopped.

This feature allows cycle-by-cycle debugging of the timer counter value, which is useful for event timing. However, in many cases, stopping the counter operation while debugging is undesirable. One example is when the timer is used in PWM mode. Stopping the timer in this case might damage the equipment that is being controlled by the PWM output.

Note that timer interrupts (if any) will not be handled before execution is resumed.

### Restore EEPROM Contents

Some AVR devices have on-chip EEPROM data memory. The ICE200 emulates the EEPROM by using an SRAM replacement inside the AVR emulator chip. This is done to eliminate problems with EEPROM write endurance. However, by doing so, a new problem is introduced because a power loss on the target board will result in loss of the data stored in the SRAM that emulates the EEPROM.

A two-step solution handles the power loss situation. First select the **Restore EEPROM** option available on the **ICE200** page. Then, before disconnecting the power, take a snapshot of the EEPROM contents by choosing **ICE200>EEPROM snapshot**. This makes the ICE200 main board read all EEPROM data into a non-volatile buffer. When power is switched off and then on again, the ICE200 will restore the contents of the buffer to the SRAM before starting code execution.

### Enable External Reset

The ATtiny12 device has a programmable fuse that lets the RESET pin function as an input pin (PB5). When this option is selected, PB5 works as a normal reset pin. When this option is deselected, PB5 works as an input pin. The device is then only reset at power-on or when giving a reset command from C-SPY. Refer to the ATtiny data sheet for more information.

---

## Resolving problems

These topics are covered:

- No contact with the target hardware
- Monitor works, but application will not run
- Optimizing downloads
- Using C-SPY macros for transparent commands
- Slow stepping speed

Debugging using the C-SPY hardware debugger systems requires interaction between many systems, independent from each other. For this reason, setting up this debug system can be a complex task. If something goes wrong, it might be difficult to locate the cause of the problem.

This section includes suggestions for resolving the most common problems that can occur when debugging with the C-SPY hardware debugger systems.

For problems concerning the operation of the evaluation board, refer to the documentation supplied with it, or contact your hardware distributor.

## NO CONTACT WITH THE TARGET HARDWARE

There are several possible reasons for C-SPY to fail to establish contact with the target hardware. Do this:

- Check the communication devices on your host computer
- Verify that the cable is properly plugged in and not damaged or of the wrong type
- Make sure that the evaluation board is supplied with sufficient power
- Check that the correct options for communication have been specified in the IAR Embedded Workbench IDE.

Examine the linker configuration file to make sure that the application has not been linked to the wrong address.

## MONITOR WORKS, BUT APPLICATION WILL NOT RUN

The application is probably linked to some illegal code area (like the interrupt table). You might have to check the defined segment allocations in the used linker configuration file.

## OPTIMIZING DOWNLOADS

C-SPY uses a special fast download mode—when the option **Fast download** is used—that runs with very little protocol overhead. The ROM-monitor must be fast enough to handle the incoming stream using full error checking on the memory or it will fail. If fast download fails, a warning message is issued by C-SPY. The download will then restart using a slower (but safer) communication protocol.

If fast download fails constantly, there are a couple of things you can do:

- Lower the transmission baud rate (if possible)
- Use RTS/CTS handshaking between C-SPY and the ROM-monitor
- Disable fast downloads.

## USING C-SPY MACROS FOR TRANSPARENT COMMANDS

You can send transparent commands directly to the ROM-monitor, using the predefined C-SPY system macro `__transparent` (*commandstring*). In this way, you can communicate directly with the ROM-monitor transparently to C-SPY.

**To execute a transparent command and set up the board for the AT90SC3232C device:**

- I Choose **View>Quick Watch** to open the Quick Watch window.

- 2 Send your transparent command to the ROM-monitor by using the predefined system macro `__transparent(commandstring)`. For this example you would type this transparent command in the text field in the **Quick Watch** window:

When you click **Recalculate**, the macro will send the macro argument `commandstring`—in this case "AT90SC3232"—as a transparent command to the Smartcard development board.

- 3 The response is displayed in the **Debug Log** window, available from the **View** menu.  
For more information about using C-SPY macros, see the chapter *Using C-SPY macros*. For information about available strings, see the documentation supplied with the evaluation board you are using.

## SLOW STEPPING SPEED

If you find that the stepping speed is slow, these troubleshooting tips might speed up stepping:

- If you are using a hardware debugger system, keep track of how many hardware breakpoints that are used and make sure some of them are left for stepping.  
Stepping in C-SPY is normally performed using breakpoints. When C-SPY performs a step command, a breakpoint is set on the next statement and the application executes until it reaches this breakpoint. If you are using a hardware debugger system, the number of hardware breakpoints—typically used for setting a stepping breakpoint in code that is located in flash/ROM memory—is limited. If you, for example, step into a C `switch` statement, breakpoints are set on each branch; this might consume several hardware breakpoints. If the number of available hardware breakpoints is exceeded, C-SPY switches into single stepping on assembly level, which can be very slow.  
For more information, see *Breakpoints in the C-SPY hardware debugger drivers*, page 116 and *Breakpoint consumers*, page 118.
- Disable trace data collection, using the **Enable/Disable** button in both the **Trace** and the **Function Profiling** windows. Trace data collection might slow down stepping because the collected trace data is processed after each step. Note that it is not sufficient to just close the corresponding windows to disable trace data collection.
- Choose to view only a limited selection of SFR registers. You can choose between two alternatives. Either type `#SFR_name` (where `SFR_name` reflects the name of the SFR you want to monitor) in the **Watch** window, or create your own filter for displaying a limited group of SFRs in the **Register** window. Displaying many SFR registers might slow down stepping because all registers must be read from the hardware after each step. See *Defining application-specific register groups*, page 144.



- Close the **Memory** and **Symbolic Memory** windows if they are open, because the visible memory must be read after each step and that might slow down stepping.
- Close any window that displays expressions such as **Watch**, **Locals**, **Statics** if it is open, because all these windows read memory after each step and that might slow down stepping.
- Close the **Stack** window if it is open. Choose **Tools>Options>Stack** and disable the **Enable graphical stack display and stack usage tracking** option if it is enabled.
- If possible, increase the communication speed between C-SPY and the target board/emulator.



## A

A access (Complex breakpoints option) . . . . . 136  
 Abort (Report Assert option) . . . . . 86  
 absolute location, specifying for a breakpoint. . . . . 138  
 Action (Data breakpoints option) . . . . . 137  
 Add to Watch Window (Symbolic Memory window context menu) . . . . . 155  
 Address Range (Find in Trace option) . . . . . 182  
 Allow download to RAM (AVR ONE! option) . . . . . 300  
 Allow download to RAM (CCR option) . . . . . 305  
 Allow download to RAM (Dragon option) . . . . . 324  
 Allow download to RAM (ICE200 option) . . . . . 308  
 Allow download to RAM (JTAGICE mkII option) . . . . . 321  
 Allow download to RAM (JTAGICE option) . . . . . 311  
 Allow download to RAM (JTAGICE3 option) . . . . . 314, 317  
 Ambiguous symbol (Resolve Symbol Ambiguity option). 106  
 application, built outside the IDE . . . . . 54  
 assembler labels, viewing . . . . . 91  
 assembler source code, fine-tuning . . . . . 183  
 assembler symbols, using in C-SPY expressions . . . . . 89  
 assembler variables, viewing. . . . . 91  
 assumptions, programming experience . . . . . 21  
 Atmel-ICE options . . . . . 312, 314  
 Atmel-ICE (C-SPY driver) . . . . . 37  
     menu . . . . . 332  
 Attach to running target (Atmel-ICE option) . . . . . 315  
 Attach to running target (JTAGICE3 option) . . . . . 318  
 Auto Scroll (Timeline window context menu) . . . . . 175  
 Auto window . . . . . 94  
 Autostep settings dialog box . . . . . 86  
 Autostep (Debug menu) . . . . . 61  
 AVR ONE! options . . . . . 298, 301  
 AVR ONE! (C-SPY driver) . . . . . 43  
     hardware installation . . . . . 44  
     menu . . . . . 333  
 --avrone\_jtag\_clock (C-SPY command line option) . . . . . 272

## B

B access (Complex breakpoints option) . . . . . 136  
 --backend (C-SPY command line option) . . . . . 272  
 backtrace information  
     generated by compiler . . . . . 74  
     viewing in Call Stack window . . . . . 80  
 batch mode, using C-SPY in . . . . . 265  
 Baud (debugger option) . . . . . 306  
 Big Endian (Memory window context menu) . . . . . 149  
 blocks, in C-SPY macros . . . . . 217  
 bold style, in this guide . . . . . 25  
 Break On Branch/Skip (Dragon menu) . . . . . 332  
 Break On Branch/Skip (JTAGICE menu) . . . . . 331  
 Break On Branch/Skip (JTAGICE mkII menu) . . . . . 331  
 Break on Throw (Debug menu) . . . . . 61  
 Break on Uncaught Exception (Debug menu) . . . . . 61  
 Break (Debug menu) . . . . . 60  
 breakpoint condition, example . . . . . 123  
 Breakpoint control (Complex breakpoints option) . . . . . 136  
 Breakpoint Usage window . . . . . 126  
 Breakpoint Usage (Atmel-ICE menu) . . . . . 333  
 Breakpoint Usage (AVR ONE! menu) . . . . . 334  
 Breakpoint Usage (CCR menu) . . . . . 335  
 Breakpoint Usage (Dragon menu) . . . . . 332  
 Breakpoint Usage (ICE200 menu) . . . . . 334  
 Breakpoint Usage (JTAGICE menu) . . . . . 331  
 Breakpoint Usage (JTAGICE mkII menu) . . . . . 331  
 Breakpoint Usage (JTAGICE3 menu) . . . . . 333  
 Breakpoint Usage (Simulator menu) . . . . . 331  
 breakpoints  
     code, example . . . . . 242  
     complex . . . . . 135  
         example . . . . . 245  
     connecting a C-SPY macro . . . . . 212  
     consumers of . . . . . 118  
     data . . . . . 130  
     data log . . . . . 132  
     description of . . . . . 113

disabling used by Stack window	119
icons for in the IDE	116
in Memory window	121
listing all	126
reasons for using	113
setting	
in memory window	121
using system macros	122
using the dialog box	120
single-stepping if not available.	52
toggling	120
types of	114
useful tips.	123
Breakpoints dialog box	
Code	127
Complex	135
Data	130
Data Log	132
Immediate	133
Log	129
Trace Start	178
Trace Stop	179
Breakpoints window	125
Browse (Trace toolbar)	169
byte order, setting in Memory window	148

## C

C function information, in C-SPY.	74
C symbols, using in C-SPY expressions	88
C variables, using in C-SPY expressions	88
call chain, displaying in C-SPY	74
Call stack information.	74
Call Stack window	80
for backtrace information.	74
Call Stack (Timeline window context menu)	176
__cancelAllInterrupts (C-SPY system macro)	223
__cancelInterrupt (C-SPY system macro).	224
CCR options	304

CCR (C-SPY driver).	48
menu	334
Chip Erase (Atmel-ICE menu)	332
Chip Erase (JTAGICE3 menu)	333
Clear All (Debug Log window context menu)	85
Clear trace data (Trace toolbar).	168
Clear (Data Log Summary window context menu)	111
Clear (Data Log window context menu)	109
__clearBreak (C-SPY system macro)	224
CLIB	
consuming breakpoints	118
documentation	23
library reference information for	24
naming convention.	26
__closeFile (C-SPY system macro)	225
code breakpoints	
overview	114
toggling	120
Code Coverage window	194
Code Coverage (Disassembly window context menu)	79
--code_coverage_file (C-SPY command line option)	272
code, covering execution of	194
command line options.	271
typographic convention	25
command prompt icon, in this guide.	25
Communication (JTAGICE mkII option)	319
complex breakpoints, overview.	115
Complex data (Complex breakpoints option)	136
computer style, typographic convention	25
conditional statements, in C-SPY macros.	216
context menu, in windows.	91
conventions, used in this guide	24
Copy Window Contents (Disassembly window context menu)	80
Copy (Debug Log window context menu)	84
copyright notice	2
--cpu (C-SPY command line option).	273
cspybat	265
reading options from file (-f)	282
current position, in C-SPY Disassembly window	78

- cursor, in C-SPY Disassembly window . . . . . 78
  - cycles (C-SPY command line option) . . . . . 273
  - C-SPY
    - batch mode, using in . . . . . 265
    - debugger systems, overview of . . . . . 33
    - differences between drivers . . . . . 35
    - environment overview . . . . . 29
    - plugin modules, loading . . . . . 53
    - setting up . . . . . 51–52
    - starting the debugger . . . . . 53
  - C-SPY drivers
    - Atmel-ICE . . . . . 37
    - AVR ONE! . . . . . 43
    - CCR . . . . . 48
    - differences between . . . . . 35
    - Dragon . . . . . 41
    - ICE200 . . . . . 47
    - JTAGICE . . . . . 45
    - JTAGICE mkII . . . . . 41
    - JTAGICE3 . . . . . 39
    - overview . . . . . 35
    - specifying . . . . . 295
    - types of . . . . . 34
  - C-SPY drivers. *See* C-SPY drivers. . . . . 48
  - C-SPY expressions . . . . . 88
    - evaluating, using Macro Quicklaunch window . . . . . 262
    - evaluating, using Quick Watch window . . . . . 102
    - in C-SPY macros . . . . . 216
    - Tooltip watch, using . . . . . 87
    - Watch window, using . . . . . 87
  - C-SPY macros
    - blocks. . . . . 217
    - conditional statements . . . . . 216
    - C-SPY expressions . . . . . 216
    - examples . . . . . 209
      - checking status of register. . . . . 211
      - creating a log macro . . . . . 212
    - executing . . . . . 209
      - connecting to a breakpoint . . . . . 212
      - using Quick Watch . . . . . 211
      - using setup macro and setup file . . . . . 211
    - functions . . . . . 89, 214
    - loop statements . . . . . 217
    - macro statements . . . . . 216
    - parameters . . . . . 215
    - setup macro file . . . . . 208
      - executing . . . . . 211
    - setup macro functions . . . . . 208
      - summary . . . . . 219
    - system macros, summary of . . . . . 221
    - using . . . . . 207
    - variables. . . . . 90, 214
  - C-SPY options
    - Extra Options . . . . . 303
    - Images . . . . . 296
    - Plugins . . . . . 297
    - Setup . . . . . 295
  - C-SPYLink . . . . . 34
  - C-STAT for static analysis, documentation for . . . . . 23
  - C++ terminology. . . . . 24
- ## D
- d (C-SPY command line option) . . . . . 274
  - Data bits (debugger option). . . . . 306
  - data breakpoints, overview . . . . . 114
  - Data Coverage (Memory window context menu) . . . . . 149
  - data coverage, in Memory window . . . . . 147
  - data log breakpoints, overview . . . . . 115
  - Data Log Summary window . . . . . 109
  - Data Log Summary (Simulator menu) . . . . . 330
  - Data Log window . . . . . 107
  - ddf (filename extension), selecting a file. . . . . 53
  - Debug Log window. . . . . 84
  - Debug menu (C-SPY main window). . . . . 60
  - Debug Port (Atmel-ICE option) . . . . . 312
  - Debug Port (AVR ONE! option) . . . . . 299
  - Debug Port (JTAGICE3 option) . . . . . 316

Debug (Report Assert option) . . . . .	86	Disassembly window . . . . .	76
--debugfile (cspybat option) . . . . .	274	context menu . . . . .	78
debugger concepts, definitions of . . . . .	32	disclaimer . . . . .	2
debugger drivers . . . . .		DLIB . . . . .	
simulator . . . . .	36	consuming breakpoints . . . . .	118
Debugger Macros window . . . . .	260	documentation . . . . .	23
debugger system overview . . . . .	33	naming convention . . . . .	26
debugging projects . . . . .		do (macro statement) . . . . .	217
externally built applications . . . . .	54	document conventions . . . . .	24
loading multiple images . . . . .	55	documentation . . . . .	
debugging, RTOS awareness . . . . .	31	overview of guides . . . . .	22
debugWIRE . . . . .	331–332	overview of this guide . . . . .	21
Default communication (JTAGICE option) . . . . .	309	Download control (Atmel-ICE option) . . . . .	313
Default communication (Serial Port option) . . . . .	306	Download control (AVR ONE! option) . . . . .	300
__delay (C-SPY system macro) . . . . .	225	Download control (CCR option) . . . . .	304
Delay (Autostep Settings option) . . . . .	86	Download control (Dragon option) . . . . .	324
Delete (Breakpoints window context menu) . . . . .	126	Download control (ICE200 option) . . . . .	308
Device description file (debugger option) . . . . .	296	Download control (JTAGICE mkII option) . . . . .	321
device description files . . . . .	53	Download control (JTAGICE option) . . . . .	310
definition of . . . . .	56	Download control (JTAGICE3 option) . . . . .	317
memory zones . . . . .	143	--download_only (C-SPY command line option) . . . . .	275
modifying . . . . .	56	Dragon options . . . . .	323, 325
register zone . . . . .	143	Dragon (C-SPY driver) . . . . .	41
Device ID (Fuse Handler option) . . . . .	337, 339	menu . . . . .	332
Devices After (AVR ONE! option) . . . . .	299	Driver (debugger option) . . . . .	295
Devices After (Dragon option) . . . . .	324	__driverType (C-SPY system macro) . . . . .	226
Devices After (JTAGICE mkII option) . . . . .	320	--drv_atmel_ice (C-SPY command line option) . . . . .	276
Devices After (JTAGICE option) . . . . .	310	--drv_communication (C-SPY command line option) . . . . .	276
Devices After (JTAGICE3 option) . . . . .	313, 317	--drv_communication_log (C-SPY command line option) . . . . .	276
Devices Before (AVR ONE! option) . . . . .	299	--drv_debug_port (C-SPY command line option) . . . . .	277
Devices Before (Dragon option) . . . . .	324	--drv_download_data (C-SPY command line option) . . . . .	277
Devices Before (JTAGICE mkII option) . . . . .	320	--drv_dragon (C-SPY command line option) . . . . .	278
Devices Before (JTAGICE option) . . . . .	310	--drv_preserve_app_section (C-SPY command line option) . . . . .	278
Devices Before (JTAGICE3 option) . . . . .	313, 317	--drv_preserve_boot_section (C-SPY command line option) . . . . .	278
Disable All (Breakpoints window context menu) . . . . .	126	--drv_set_exit_breakpoint (C-SPY command line option) . . . . .	279
Disable (Breakpoints window context menu) . . . . .	126	--drv_set_getchar_breakpoint (C-SPY command line option) . . . . .	279
__disableInterrupts (C-SPY system macro) . . . . .	225		
--disable_internal_eeprom (C-SPY command line option) . . . . .	275		
--disable_interrupts (C-SPY command line option) . . . . .	275		

--drv\_set\_putchar\_breakpoint  
(C-SPY command line option) . . . . . 280  
 --drv\_suppress\_download (C-SPY command line option) 280  
 --drv\_use\_PDI (C-SPY command line option) . . . . . 281  
 --drv\_verify\_download (C-SPY command line option) . . 281

## E

Edit Expressions (Trace toolbar) . . . . . 169  
 Edit Settings (Trace toolbar) . . . . . 169  
 Edit (Breakpoints window context menu) . . . . . 125  
 edition, of this guide . . . . . 2  
 EEPROM  
   contents, preserving in Atmel-ICE . . . . . 314  
   contents, preserving in AVR ONE! . . . . . 301  
   contents, preserving in Dragon . . . . . 325  
   contents, preserving in JTAGICE . . . . . 311  
   contents, preserving in JTAGICE mkII . . . . . 322  
   contents, preserving in JTAGICE3 . . . . . 318  
 --eeprom\_size (C-SPY command line option) . . . . . 281  
 Embedded C++ Technical Committee . . . . . 24  
 Enable All (Breakpoints window context menu) . . . . . 126  
 Enable External Reset (ICE200 Options option) . . . . . 342  
 Enable interrupt simulation (Interrupt Setup option) . . . . 204  
 Enable Log File (Log File option) . . . . . 85  
 Enable software breakpoints (Atmel-ICE option) . . . . . 315  
 Enable software breakpoints (AVR ONE! option) . . . . . 301  
 Enable software breakpoints (Dragon option) . . . . . 325  
 Enable software breakpoints (JTAGICE mkII option) . . . . . 322  
 Enable software breakpoints (JTAGICE3 option) . . . . . 318  
 Enable (Breakpoints window context menu) . . . . . 126  
 Enable (Data Log Summary window context menu) . . . . . 111  
 Enable (Data Log window context menu) . . . . . 109  
 Enable (Timeline window context menu) . . . . . 176  
 \_\_enableInterrupts (C-SPY system macro) . . . . . 226  
 Enable/Disable Breakpoint (Call  
 Stack window context menu) . . . . . 82  
 Enable/Disable Breakpoint (Disassembly window context  
 menu) . . . . . 80  
 Enable/Disable (Trace toolbar) . . . . . 168

endianness. *See* byte order  
 Ending Session Disables debugWire (AVR ONE! menu) . 334  
 Ending Session Disables debugWIRE (Dragon menu) . . . 332  
 Ending Session Disables  
 debugWIRE (JTAGICE mkII menu) . . . . . 331  
 --enhanced\_core (C-SPY command line option) . . . . . 282  
 Enter Location dialog box . . . . . 138  
 \_\_evaluate (C-SPY system macro) . . . . . 227  
 Evaluate Now (Macro Quicklaunch  
 window context menu) . . . . . 263  
 examples  
   C-SPY macros . . . . . 209  
   interrupts  
     timer . . . . . 202  
   macros  
     checking status of register . . . . . 211  
     creating a log macro . . . . . 212  
     using Quick Watch . . . . . 211  
     performing tasks and continue execution . . . . . 123  
     tracing incorrect function arguments . . . . . 123  
 execUserExecutionStarted (C-SPY setup macro) . . . . . 220  
 execUserExecutionStopped (C-SPY setup macro) . . . . . 220  
 execUserExit (C-SPY setup macro) . . . . . 221  
 execUserPreload (C-SPY setup macro) . . . . . 219  
 execUserPreReset (C-SPY setup macro) . . . . . 221  
 execUserReset (C-SPY setup macro) . . . . . 221  
 execUserSetup (C-SPY setup macro) . . . . . 220  
 executed code, covering . . . . . 194  
 execution history, tracing . . . . . 167  
 expressions. *See* C-SPY expressions  
 extended command line file, for cspybat . . . . . 282  
 Extra Options, for C-SPY . . . . . 303

## F

-f (cspybat option) . . . . . 282  
 File format (Memory Save option) . . . . . 150  
 file types  
   device description, specifying in IDE . . . . . 53

macro	52, 295
filename extensions	
ddf, selecting device description file	53
mac, using macro file	52
Filename (Memory Restore option)	151
Filename (Memory Save option)	151
Fill dialog box	152
__writeMemory8 (C-SPY system macro)	227
__writeMemory16 (C-SPY system macro)	228
__writeMemory32 (C-SPY system macro)	229
Find in Trace dialog box	181
Find in Trace window	182
Find (Memory window context menu)	149
Find (Trace toolbar)	169
first activation time (interrupt property)	
definition of	200
flash memory, load library module to	232
for (macro statement)	217
formats, C-SPY input	31
Frequency in Hz (AVR ONE! option)	299
Frequency in Hz (JTAGICE mkII option)	320, 323
Frequency in Hz (JTAGICE option)	310
Frequency in Hz (JTAGICE3 option)	313, 316
Function Profiler window	188
Function Profiler (Simulator menu)	330
Function Trace window	170
Function Trace (Simulator menu)	330
functions	
call stack information for	74
C-SPY running to when starting	52, 295
most time spent in, locating	183
Fuse Handler dialog box	335, 338
Fuse Handler (Atmel-ICE menu)	332
Fuse Handler (AVR ONE! menu)	333
Fuse Handler (Dragon menu)	332
Fuse Handler (JTAGICE mkII menu)	331
Fuse Handler (JTAGICE3 menu)	333

## G

Get Example Projects dialog box	66
__getCycleCounter (C-SPY system macro)	230
Go to Source (Breakpoints window context menu)	125
Go to Source (Call Stack window context menu)	81
Go To Source (Timeline window context menu)	176
Go (Debug menu)	60, 73

## H

Handshaking (debugger option)	306
Hardware reset on C-SPY reset (AVR ONE! option)	301
Hardware reset on C-SPY reset (Dragon option)	325
Hardware reset on C-SPY reset (JTAGICE mkII option)	322
Hardware reset on C-SPY reset (JTAGICE option)	312
High speed (ICE200 option)	307
highlighting, in C-SPY	74

## I

IAR debugger driver plugin (debugger option)	326
ICE200 options	307
ICE200 Options dialog box	340
ICE200 (C-SPY driver)	47
hardware installation	48
menu	334
--ice200_restore_EEPROM (C-SPY command line option)	282
--ice200_single_step_timers (C-SPY command line option)	283
icons, in this guide	25
if else (macro statement)	216
if (macro statement)	216
Ignore (Report Assert option)	86
Images window	64
Images, loading multiple	296
immediate breakpoints, overview	115



Include (Log File option) . . . . . 85  
 input formats, C-SPY . . . . . 31  
 Input Mode dialog box . . . . . 83  
 input, special characters in Terminal I/O window . . . . . 83  
 installation directory . . . . . 24  
 Instruction bits After (AVR ONE! option) . . . . . 299  
 Instruction bits After (Dragon option) . . . . . 324  
 Instruction bits After (JTAGICE mkII option) . . . . . 320  
 Instruction bits After (JTAGICE option) . . . . . 310  
 Instruction bits After (JTAGICE3 option) . . . . . 313, 317  
 Instruction bits Before (AVR ONE! option) . . . . . 299  
 Instruction bits Before (Dragon option) . . . . . 324  
 Instruction bits Before (JTAGICE mkII option) . . . . . 320  
 Instruction bits Before (JTAGICE option) . . . . . 310  
 Instruction bits Before (JTAGICE3 option) . . . . . 313, 317  
 Instruction Profiling (Disassembly window context menu) . 79  
 Intel-extended, C-SPY input format . . . . . 31  
 Intel-extended, C-SPY output format . . . . . 34  
 Interrupt Log Summary (Simulator menu) . . . . . 330  
 interrupt system, using device description file . . . . . 201  
 interrupts  
     adapting C-SPY system for target hardware . . . . . 201  
     simulated, introduction to . . . . . 199  
     timer, example . . . . . 202  
     using system macros . . . . . 201  
 Interrupts dialog box . . . . . 204  
 \_\_isBatchMode (C-SPY system macro) . . . . . 230  
 italic style, in this guide . . . . . 25  
 I/O register. *See* SFR

## J

JTAG daisy chain (AVR ONE! option) . . . . . 299  
 JTAG daisy chain (Dragon option) . . . . . 323  
 JTAG daisy chain (JTAGICE mkII option) . . . . . 320  
 JTAG daisy chain (JTAGICE option) . . . . . 310  
 JTAG daisy chain (JTAGICE3 option) . . . . . 316  
 JTAG Port (Atmel-ICE option) . . . . . 313  
 JTAG Port (AVR ONE! option) . . . . . 299

JTAG Port (Dragon option) . . . . . 323  
 JTAG Port (JTAGICE mkII option) . . . . . 320  
 JTAG Port (JTAGICE option) . . . . . 310  
 JTAG Port (JTAGICE3 option) . . . . . 316  
 JTAGICE mkII options . . . . . 319, 322  
 JTAGICE mkII (C-SPY driver) . . . . . 41  
     hardware installation . . . . . 42  
     menu . . . . . 331  
 JTAGICE options . . . . . 309, 311  
 JTAGICE (C-SPY driver) . . . . . 45  
     menu . . . . . 331  
 --jtagicemkII\_use\_software\_breakpoints (C-SPY command  
 line option) . . . . . 285  
 --jtagice\_clock (C-SPY command line option) . . . . . 283  
 --jtagice\_do\_hardware\_reset  
 (C-SPY command line option) . . . . . 283  
 --jtagice\_leave\_timers\_running  
 (C-SPY command line option) . . . . . 284  
 --jtagice\_preserve\_eeprom (C-SPY command line option) 284  
 --jtagice\_restore\_fuse (C-SPY command line option) . . . 285  
 JTAGICE3 options . . . . . 316, 318  
 JTAGICE3 (C-SPY driver) . . . . . 39  
     hardware installation . . . . . 38, 40  
     menu . . . . . 333

## L

labels (assembler), viewing . . . . . 91  
 Length (Fill option) . . . . . 152  
 library functions  
     C-SPY support for using, plugin module . . . . . 287  
     online help for . . . . . 24  
 lightbulb icon, in this guide . . . . . 25  
 linker options  
     typographic convention . . . . . 25  
     consuming breakpoints . . . . . 118  
 Little Endian (Memory window context menu) . . . . . 148  
 \_\_loadImage (C-SPY system macro) . . . . . 231  
 loading multiple debug files, list currently loaded . . . . . 64  
 loading multiple images . . . . . 55

Locals window .....	95
log breakpoints, overview .....	114
Log communication (debugger option) .....	306, 327
Log File dialog box .....	85
Log Messages (Fuse Handler option) .....	337, 340
Logging>Set Log file (Debug menu) .....	62
Logging>Set Terminal I/O Log file (Debug menu) .....	62
loop statements, in C-SPY macros .....	217

## M

mac (filename extension), using a macro file .....	52
--macro (C-SPY command line option) .....	286
macro files, specifying .....	52, 295
Macro Quicklaunch window .....	262
Macro Registration window .....	258
macro statements .....	216
macros	
executing .....	209
using .....	207
Macros (Debug menu) .....	62
--macro-param (C-SPY command line option) .....	286
main function, C-SPY running to when starting .....	52, 295
Memory Fill (Memory window context menu) .....	149
Memory Restore dialog box .....	151
Memory Restore (Memory window context menu) .....	149
Memory Save dialog box .....	150
Memory Save (Memory window context menu) .....	149
Memory window .....	146
memory zones .....	142
in device description file .....	143
__memoryRestore (C-SPY system macro) .....	232
__memoryRestoreFromFile (C-SPY system macro) .....	233
__memorySave (C-SPY system macro) .....	233
__memorySaveToFile (C-SPY system macro) .....	234
Memory>Restore (Debug menu) .....	62
Memory>Save (Debug menu) .....	62
menu bar, C-SPY-specific .....	59
__messageBoxYesCancel (C-SPY system macro) .....	235

__messageBoxYesNo (C-SPY system macro) .....	235
migration	
from earlier IAR compilers .....	23
MISRA C, documentation .....	23
Mixed Mode (Disassembly window context menu) .....	80
Motorola, C-SPY input format .....	31
Motorola, C-SPY output format .....	34
Move to PC (Disassembly window context menu) .....	78

## N

naming conventions .....	25
Navigate (Timeline window context menu) .....	175
New Breakpoint (Breakpoints window context menu) .....	126
Next Statement (Debug menu) .....	61
Next Symbol (Symbolic Memory window context menu) .....	155

## O

__openFile (C-SPY system macro) .....	236
Operation (Fill option) .....	152
operators, sizeof in C-SPY .....	90
optimizations, effects on variables .....	90
options	
in the IDE .....	293
on the command line .....	271, 303
Options (Stack window context menu) .....	159
__orderInterrupt (C-SPY system macro) .....	237
Originator (debugger option) .....	297

## P

-p (C-SPY command line option) .....	287
parameters	
tracing incorrect values of .....	74
typographic convention .....	25
Parity (debugger option) .....	306
part number, of this guide .....	2

- peripheral units
    - device-specific . . . . . 56
    - displayed in Register window . . . . . 142
    - in C-SPY expressions . . . . . 89
    - initializing using setup macros . . . . . 208
  - peripherals register. *See* SFR
  - Please select one symbol
    - (Resolve Symbol Ambiguity option) . . . . . 106
    - plugin (C-SPY command line option) . . . . . 287
  - plugin modules (C-SPY). . . . . 34
    - loading . . . . . 53
  - Plugins (C-SPY options). . . . . 297
  - pop-up menu. *See* context menu
  - Port (Serial Port option) . . . . . 306
  - prerequisites, programming experience. . . . . 21
  - Preserve EEPROM contents (Atmel-ICE option) . . . . . 314
  - Preserve EEPROM contents (AVR ONE! option) . . . . . 301
  - Preserve EEPROM contents (Dragon option) . . . . . 325
  - Preserve EEPROM contents (JTAGICE mkII option) . . . . . 322
  - Preserve EEPROM contents (JTAGICE3 option) . . . . . 311
  - Preserve EEPROM contents (JTAGICE3 option) . . . . . 318
  - Preserve FLASH (Atmel-ICE option) . . . . . 315
  - Preserve FLASH (AVR ONE! option) . . . . . 302
  - Preserve FLASH (JTAGICE mkII option) . . . . . 323
  - Preserve FLASH (JTAGICE3 option) . . . . . 319
  - Previous Symbol (Symbolic Memory window context menu) . . . . . 155
  - probability (interrupt property)
    - definition of . . . . . 200
  - Profile Selection (Timeline window context menu) . . . . . 176
  - profiling
    - analyzing data . . . . . 185
      - on function level . . . . . 184
      - on instruction level. . . . . 187
  - profiling information, on functions and instructions . . . . . 183
  - profiling sources
    - trace (calls) . . . . . 184, 189
    - trace (flat) . . . . . 184, 190
  - program execution
    - breaking . . . . . 114–115
      - in C-SPY . . . . . 69
    - Program Fuses (Fuse Handler option) . . . . . 337, 339
    - programming experience. . . . . 21
    - program. *See* application
    - projects, for debugging externally built applications. . . . . 54
    - publication date, of this guide. . . . . 2
- ## Q
- Quick Watch window . . . . . 102
    - executing C-SPY macros . . . . . 211
- ## R
- Range for (Viewing Range option) . . . . . 177
  - Read Fuses (Fuse Handler option) . . . . . 337, 339
  - \_\_readFile (C-SPY system macro) . . . . . 238
  - \_\_readFileByte (C-SPY system macro) . . . . . 239
  - \_\_readMemoryByte (C-SPY system macro) . . . . . 239
  - \_\_readMemory8 (C-SPY system macro) . . . . . 239
  - \_\_readMemory16 (C-SPY system macro) . . . . . 240
  - \_\_readMemory32 (C-SPY system macro) . . . . . 240
  - reference information, typographic convention. . . . . 25
  - Refresh (Debug menu) . . . . . 62
  - register groups . . . . . 142
    - predefined, enabling. . . . . 160
  - Register window . . . . . 159
  - registered trademarks . . . . . 2
  - \_\_registerMacroFile (C-SPY system macro) . . . . . 241
  - registers, displayed in Register window . . . . . 159
  - Remove All (Macro Quicklaunch window context menu) . . . . . 263
  - Remove (Macro Quicklaunch window context menu) . . . . . 263
  - repeat interval (interrupt property), definition of. . . . . 200
  - Replace (Memory window context menu) . . . . . 149
  - Report Assert dialog box . . . . . 86
  - Reset Delay (ICE200 Options option) . . . . . 340
  - Reset (Debug menu) . . . . . 60
  - \_\_resetFile (C-SPY system macro) . . . . . 241

Resolve Source Ambiguity dialog box	139
Restore EEPROM Contents (ICE200 Options option)	342
Restore EEPROM (ICE200 option)	307
Restore fuses when ending debug session (Atmel-ICE option)	315
Restore fuses when ending debug session (AVR ONE! option)	301
Restore fuses when ending debug session (JTAGICE option)	312
Restore fuses when ending debug session (JTAGICE3 option)	318
Restore (Memory Restore option)	151
return (macro statement)	217
ROM-monitor, definition of	34
RTOS awareness debugging	31
RTOS awareness (C-SPY plugin module)	31
Run timers in stopped mode (Atmel-ICE option)	314
Run timers in stopped mode (AVR ONE! option)	301
Run timers in stopped mode (Dragon option)	325
Run timers in stopped mode (JTAGICE mkII option)	322
Run timers in stopped mode (JTAGICE option)	311
Run timers in stopped mode (JTAGICE3 option)	318
Run to Cursor (Call Stack window context menu)	81
Run to Cursor (Debug menu)	61
Run to Cursor (Disassembly window context menu)	79
Run to Cursor, command for executing	74
Run to (C-SPY option)	52, 295

## S

Save to log file (context menu command)	109, 111
Save (Memory Save option)	151
Save (Trace toolbar)	169
Scale (Viewing Range option)	177
Select All (Debug Log window context menu)	84
Select Graphs (Timeline window context menu)	176
Select plugins to load (debugger option)	297
serial port setup, hardware drivers	305
Set Data Breakpoint (Memory window context menu)	149

Set Data Log	
Breakpoint (Memory window context menu)	150
Set Next Statement (Debug menu)	61
Set Next Statement (Disassembly window context menu)	80
__setCodeBreak (C-SPY system macro)	242
__setComplexBreak (C-SPY system macro)	243
__setDataBreak (C-SPY system macro)	245
__setLogBreak (C-SPY system macro)	247
__setSimBreak (C-SPY system macro)	248
__setTraceStartBreak (C-SPY system macro)	249
__setTraceStopBreak (C-SPY system macro)	250
setup macro file, registering	52
setup macro functions	208
reserved names	219
Setup macros (debugger option)	295
Setup (C-SPY options)	295
SFR	
in Register window	160
using as assembler symbols	89
shortcut menu. <i>See</i> context menu	
Show all images (Images window context menu)	65
Show Arguments (Call Stack window context menu)	81
Show Cycles (Data Log Summary window context menu)	111
Show Cycles (Data Log window context menu)	109
Show offsets (Stack window context menu)	158
Show only (Image window context menu)	65
Show Time (Data Log Summary window context menu)	111
Show Time (Data Log window context menu)	109
Show variables (Stack window context menu)	158
--silent (C-SPY command line option)	288
simulating interrupts, enabling/disabling	204
Simulator menu	330
simulator, introduction	36
Single Step Timers (ICE200 Options option)	341
Single step timers (ICE200 option)	308
sizeof	90
software breakpoints	
enabling for Atmel-ICE	315
enabling for AVR Dragon	285, 325
enabling for AVR ONE!	285, 301

- enabling for JTAGICE mkII . . . . . 285, 322
- enabling for JTAGICE3 . . . . . 285, 318
- use of . . . . . 118
- \_\_sourcePosition (C-SPY system macro) . . . . . 251
- special function registers (SFR)
  - in Register window . . . . . 160
  - using as assembler symbols . . . . . 89
- stack usage, computing . . . . . 143
- Stack window . . . . . 156
- stack.mac . . . . . 207
- standard C, sizeof operator in C-SPY . . . . . 90
- Start address (Fill option) . . . . . 152
- Start address (Memory Save option) . . . . . 150
- static analysis
  - documentation for . . . . . 23
- Statics window . . . . . 99
- stdin and stdout, redirecting to C-SPY window . . . . . 82
- Step Into (Debug menu) . . . . . 61
- Step Into, description . . . . . 71
- Step Out (Debug menu) . . . . . 61
- Step Out, description . . . . . 72
- Step Over (Debug menu) . . . . . 61
- Step Over, description . . . . . 71
- step points, definition of . . . . . 70
- Stop address (Memory Save option) . . . . . 150
- Stop bits (debugger option) . . . . . 306
- Stop Debugging (Debug menu) . . . . . 61
- \_\_strFind (C-SPY system macro) . . . . . 251
- \_\_subString (C-SPY system macro) . . . . . 252
- Suppress download
  - AVR ONE! option . . . . . 300
  - CCR option . . . . . 305
  - debugger option . . . . . 326
  - Dragon option . . . . . 324
  - ICE200 option . . . . . 308
  - JTAGICE mkII option . . . . . 321
  - JTAGICE option . . . . . 310
  - JTAGICE3 option . . . . . 313, 317
- Symbolic Memory window . . . . . 153

- Symbols window . . . . . 104
- symbols, using in C-SPY expressions . . . . . 88
- System breakpoints on (Atmel-ICE option) . . . . . 315
- System breakpoints on (AVR ONE! option) . . . . . 301
- System breakpoints on (Dragon option) . . . . . 325
- System breakpoints on (JTAGICE mkII option) . . . . . 322
- System breakpoints on (JTAGICE option) . . . . . 312
- System breakpoints on (JTAGICE3 option) . . . . . 318

## T

- Target Consistency Check (AVR ONE! option) . . . . . 300
- Target Consistency Check (CCR option) . . . . . 305
- Target Consistency Check (Dragon option) . . . . . 324
- Target Consistency Check (ICE200 option) . . . . . 309
- Target Consistency Check (JTAGICE mkII option) . . . . . 320–321
- Target Consistency Check (JTAGICE option) . . . . . 311
- Target Consistency Check (JTAGICE3 option) . . . . . 314, 317
- Target device is part of a JTAG
  - daisy chain (AVR ONE! option) . . . . . 299
  - Target device is part of a JTAG
    - daisy chain (Dragon option) . . . . . 323
    - Target device is part of a JTAG
      - daisy chain (JTAGICE mkII option) . . . . . 320
      - Target device is part of a JTAG
        - daisy chain (JTAGICE option) . . . . . 310
        - Target device is part of a JTAG
          - daisy chain (JTAGICE3 option) . . . . . 313, 316
  - target system, definition of . . . . . 33
  - \_\_targetDebuggerVersion (C-SPY system macro) . . . . . 252
  - Terminal IO Log Files (Terminal IO Log Files option) . . . . . 84
  - Terminal I/O Log Files dialog box . . . . . 83
  - Terminal I/O window . . . . . 75, 82
  - terminology . . . . . 24
  - Text search (Find in Trace option) . . . . . 181
  - Third-Party Driver (debugger options) . . . . . 326
  - Time Axis Unit (Timeline window context menu) . . . . . 176
  - Timeline window . . . . . 171
  - Timeline (Simulator menu) . . . . . 330

--timeout (C-SPY command line option) . . . . .	288
timer interrupt, example . . . . .	202
timers (Atmel-ICE), running in stopped mode . . . . .	314
timers (AVR ONE!), running in stopped mode . . . . .	301
timers (Dragon), running in stopped mode . . . . .	325
timers (JTAGICE mkII), running in stopped mode . . . . .	322
timers (JTAGICE), running in stopped mode . . . . .	311
timers (JTAGICE3), running in stopped mode . . . . .	318
Toggle Breakpoint (Code) (Call Stack window context menu) . . . . .	81
Toggle Breakpoint (Code) (Disassembly window context menu) . . . . .	79
Toggle Breakpoint (Log) (Call Stack window context menu) . . . . .	82
Toggle Breakpoint (Log) (Disassembly window context menu) . . . . .	79
Toggle Breakpoint (Trace Start) (Call Stack window context menu) . . . . .	82
Toggle Breakpoint (Trace Start) (Disassembly window context menu) . . . . .	79
Toggle Breakpoint (Trace Stop) (Call Stack window context menu) . . . . .	82
Toggle Breakpoint (Trace Stop) (Disassembly window context menu) . . . . .	80
Toggle source (Trace toolbar) . . . . .	169
__toLower (C-SPY system macro) . . . . .	253
tools icon, in this guide . . . . .	25
__toString (C-SPY system macro) . . . . .	253
__toUpper (C-SPY system macro) . . . . .	254
trace . . . . .	165
in Timeline window . . . . .	171
Trace Expressions window . . . . .	180
trace start and stop breakpoints, overview . . . . .	114
Trace Start breakpoints dialog box . . . . .	178
Trace Stop breakpoints dialog box . . . . .	179
Trace window . . . . .	168
trace (calls), profiling source . . . . .	184, 189
trace (flat), profiling source . . . . .	184, 190
Trace (Simulator menu) . . . . .	330
trademarks . . . . .	2

__transparent (C-SPY system macro) . . . . .	254
typographic conventions . . . . .	25

## U

UBROF . . . . .	31
Unavailable, C-SPY message . . . . .	91
Universal Binary Relocatable Object Format. <i>See</i> UBROF	
__unloadImage(C-SPY system macro) . . . . .	255
USB . . . . .	
AVR ONE! option . . . . .	302
JTAGICE mkII option . . . . .	319
USB ID (AVR ONE! option) . . . . .	302
USB ID (JTAGICE mkII option) . . . . .	319
Use command line options (debugger option) . . . . .	303
Use Extra Images (debugger option) . . . . .	296
Use PDI (JTAGICE mkII option) . . . . .	319, 323
Use UBROF reset vector (debugger option) . . . . .	295
user application, definition of . . . . .	33

## V

-v (C-SPY command line option) . . . . .	288
Value (Fill option) . . . . .	152
variables . . . . .	
effects of optimizations . . . . .	90
information, limitation on . . . . .	90
using in C-SPY expressions . . . . .	88
variance (interrupt property), definition of . . . . .	200
Verify all (debugger option) . . . . .	326
version . . . . .	
of this guide . . . . .	2
Viewing Range dialog box . . . . .	177
visualSTATE, C-SPY plugin module for . . . . .	34

## W

warnings icon, in this guide . . . . .	25
--	----

Watch window	97
using	87
web sites, recommended	24
while (macro statement)	217
windows, specific to C-SPY	62
With I/O emulation modules (linker option), using	82
__writeFile (C-SPY system macro)	255
__writeFileByte (C-SPY system macro)	256
__writeMemoryByte (C-SPY system macro)	256
__writeMemory8 (C-SPY system macro)	256
__writeMemory16 (C-SPY system macro)	257
__writeMemory32 (C-SPY system macro)	257

## Z

zone	
defined in device description file	143
in C-SPY	142
part of an absolute address	138
Zoom (Timeline window context menu)	175

## Symbols

__cancelAllInterrupts (C-SPY system macro)	223
__cancelInterrupt (C-SPY system macro)	224
__clearBreak (C-SPY system macro)	224
__closeFile (C-SPY system macro)	225
__delay (C-SPY system macro)	225
__disableInterrupts (C-SPY system macro)	225
__driverType (C-SPY system macro)	226
__enableInterrupts (C-SPY system macro)	226
__evaluate (C-SPY system macro)	227
__fillMemory8 (C-SPY system macro)	227
__fillMemory16 (C-SPY system macro)	228
__fillMemory32 (C-SPY system macro)	229
__fmessage (C-SPY macro statement)	217
__getCycleCounter (C-SPY system macro)	230
__isBatchMode (C-SPY system macro)	230
__loadImage (C-SPY system macro)	231

__memoryRestore (C-SPY system macro)	232
__memoryRestoreFromFile (C-SPY system macro)	233
__memorySave (C-SPY system macro)	233
__memorySaveToFile (C-SPY system macro)	234
__message (C-SPY macro statement)	217
__messageBoxYesCancel (C-SPY system macro)	235
__messageBoxYesNo (C-SPY system macro)	235
__openFile (C-SPY system macro)	236
__orderInterrupt (C-SPY system macro)	237
__readFile (C-SPY system macro)	238
__readFileByte (C-SPY system macro)	239
__readMemoryByte (C-SPY system macro)	239
__readMemory8 (C-SPY system macro)	239
__readMemory16 (C-SPY system macro)	240
__readMemory32 (C-SPY system macro)	240
__registerMacroFile (C-SPY system macro)	241
__resetFile (C-SPY system macro)	241
__setCodeBreak (C-SPY system macro)	242
__setComplexBreak (C-SPY system macro)	243
__setDataBreak (C-SPY system macro)	245
__setLogBreak (C-SPY system macro)	247
__setSimBreak (C-SPY system macro)	248
__setTraceStartBreak (C-SPY system macro)	249
__setTraceStopBreak (C-SPY system macro)	250
__smessage (C-SPY macro statement)	217
__sourcePosition (C-SPY system macro)	251
__strFind (C-SPY system macro)	251
__subString (C-SPY system macro)	252
__targetDebuggerVersion (C-SPY system macro)	252
__toLowerCase (C-SPY system macro)	253
__toString (C-SPY system macro)	253
__toUpper (C-SPY system macro)	254
__transparent (C-SPY system macro)	254
__unloadImage (C-SPY system macro)	255
__writeFile (C-SPY system macro)	255
__writeFileByte (C-SPY system macro)	256
__writeMemoryByte (C-SPY system macro)	256
__writeMemory8 (C-SPY system macro)	256
__writeMemory16 (C-SPY system macro)	257

__writeMemory32 (C-SPY system macro) . . . . .	257
-d (C-SPY command line option) . . . . .	274
-f (cspybat option) . . . . .	282
-p (C-SPY command line option) . . . . .	287
-v (C-SPY command line option) . . . . .	288
--avrone_jtag_clock (C-SPY command line option) . . . . .	272
--backend (C-SPY command line option) . . . . .	272
--code_coverage_file (C-SPY command line option) . . . . .	272
--cpu (C-SPY command line option) . . . . .	273
--cycles (C-SPY command line option) . . . . .	273
--debugfile (cspybat option) . . . . .	274
--disable_internal_eeeprom (C-SPY command line option) . . . . .	275
--disable_interrupts (C-SPY command line option) . . . . .	275
--download_only (C-SPY command line option) . . . . .	275
--drv_atmel_ice (C-SPY command line option) . . . . .	276
--drv_communication (C-SPY command line option) . . . . .	276
--drv_communication_log (C-SPY command line option) . . . . .	276
--drv_debug_port (C-SPY command line option) . . . . .	277
--drv_download_data (C-SPY command line option) . . . . .	277
--drv_dragon (C-SPY command line option) . . . . .	278
--drv_preserve_app_section (C-SPY command line option) . . . . .	278
--drv_preserve_boot_section (C-SPY command line option) . . . . .	278
--drv_set_exit_breakpoint (C-SPY command line option) . . . . .	279
--drv_set_getchar_breakpoint (C-SPY command line option) . . . . .	279
--drv_set_putchar_breakpoint (C-SPY command line option) . . . . .	280
--drv_suppress_download (C-SPY command line option) . . . . .	280
--drv_use_PDI (C-SPY command line option) . . . . .	281
--drv_verify_download (C-SPY command line option) . . . . .	281
--eeeprom_size (C-SPY command line option) . . . . .	281
--enhanced_core (C-SPY command line option) . . . . .	282
--ice200_restore_EEPROM (C-SPY command line option) . . . . .	282

--ice200_single_step_timers (C-SPY command line option) . . . . .	283
--jtagicemkII_use_software_breakpoints (C-SPY command line option) . . . . .	285
--jtagice_clock (C-SPY command line option) . . . . .	283
--jtagice_do_hardware_reset (C-SPY command line option) . . . . .	283
--jtagice_leave_timers_running (C-SPY command line option) . . . . .	284
--jtagice_preserve_eeeprom (C-SPY command line option) . . . . .	284
--jtagice_restore_fuse (C-SPY command line option) . . . . .	285
--macro (C-SPY command line option) . . . . .	286
--macro_param (C-SPY command line option) . . . . .	286
--plugin (C-SPY command line option) . . . . .	287
--silent (C-SPY command line option) . . . . .	288
--timeout (C-SPY command line option) . . . . .	288
--64bit_doubles (C-SPY command line option) . . . . .	271
--64k_flash (C-SPY command line option) . . . . .	271

## Numerics

1x Units (Symbolic Memory window context menu) . . . . .	155
--64bit_doubles (C-SPY command line option) . . . . .	271
--64k_flash (C-SPY command line option) . . . . .	271
8x Units (Memory window context menu) . . . . .	148