

IAR Embedded Workbench[®]

C-SPY[®] Debugging Guide

for Atmel[®] Corporation's
AVR Microcontroller Family



UCSAVR-I

 **IAR**
SYSTEMS

COPYRIGHT NOTICE

Copyright © 2011 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, visualSTATE, From Idea To Target, IAR KickStart Kit, IAR PowerPac, IAR YellowSuite, IAR Advanced Development Kit, IAR, and the IAR Systems logotype are trademarks or registered trademarks owned by IAR Systems AB. J-Link is a trademark licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Atmel is a registered trademark of Atmel® Corporation. AVR is a trademark 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

First edition: June 2011

Part number: UCSAVR-1

This guide applies to version 6.1x of IAR Embedded Workbench® for AVR.

The *C-SPY® Debugging Guide for AVR* replaces all debugging information in the *IAR Embedded Workbench IDE User Guide for AVR* and the hardware debugger guides for AVR.

Internal reference: M10, Too6.3, IMAE.

Brief contents

Tables	15
Figures	17
Preface	21
The IAR C-SPY Debugger	27
Getting started using C-SPY	47
Executing your application	63
Working with variables and expressions	81
Using breakpoints	95
Monitoring memory and registers	121
Collecting and using trace data	139
Using the profiler	153
Code coverage	161
Interrupts	165
Using C-SPY macros	173
The C-SPY Command Line Utility—cspybat	219
Debugger options	237
Additional information on C-SPY drivers	269
Index	287

Contents

Tables	15
Figures	17
Preface	21
Who should read this guide	21
How to use this guide	21
What this guide contains	22
Other documentation	23
User and reference guides	23
The online help system	24
Web sites	24
Document conventions	24
Typographic conventions	25
Naming conventions	25
The IAR C-SPY Debugger	27
Introduction to C-SPY	27
An integrated environment	27
General C-SPY debugger features	28
RTOS awareness	29
Debugger concepts	30
C-SPY and target systems	30
The debugger	31
The target system	31
The application	31
C-SPY debugger systems	32
The ROM-monitor program	32
Third-party debuggers	32
C-SPY plugin modules	32
C-SPY drivers overview	33
Differences between the C-SPY drivers	33

The IAR C-SPY Simulator	34
Features	34
Selecting the simulator driver	34
The C-SPY JTAGICE driver	35
Features	35
Communication overview	36
Hardware installation	36
The C-SPY JTAGICE mkII/Dragon driver	37
Features	37
Communication overview	38
Hardware installation	38
The C-SPY JTAGICE3 driver	39
Features	39
Communication overview	40
Hardware installation	40
The C-SPY AVR ONE! driver	41
Features	41
Communication overview	42
Hardware installation	42
The C-SPY ICE200 driver	43
Features	43
Communication overview	44
Supported devices	44
Hardware installation	44
Upgrading the firmware	45
The C-SPY Crypto Controller ROM-monitor driver	45
Features	45
Communication overview	46
Getting started using C-SPY	47
Setting up C-SPY	47
Setting up for debugging	47
Executing from reset	48
Using a setup macro file	48

Selecting a device description file	49
Loading plugin modules	49
Starting C-SPY	49
Starting the debugger	50
Loading executable files built outside of the IDE	50
Starting a debug session with source files missing	50
Loading multiple images	51
Adapting for target hardware	52
Modifying a device description file	52
Initializing target hardware before C-SPY starts	53
Running example projects	53
Running an example project	54
Reference information on starting C-SPY	55
C-SPY Debugger main window	55
Images window	60
Get Alternative File dialog box	61
Executing your application	63
Introduction to application execution	63
Briefly about application execution	63
Source and disassembly mode debugging	63
Single stepping	64
Running the application	66
Highlighting	66
Call stack information	67
Terminal input and output	68
Debug logging	68
Reference information on application execution	68
Disassembly window	69
Call Stack window	73
Terminal I/O window	74
Terminal I/O Log File dialog box	75
Debug Log window	76
Log File dialog box	77

Report Assert dialog box	78
Autostep settings dialog box	78
Working with variables and expressions	81
Introduction to working with variables and expressions	81
Briefly about working with variables and expressions	81
C-SPY expressions	82
Limitations on variable information	84
Viewing assembler variables	85
Reference information on working with variables and expressions	86
Auto window	86
Locals window	87
Watch window	87
Statics window	89
Quick Watch window	92
Symbols window	93
Resolve Symbol Ambiguity dialog box	94
Using breakpoints	95
Introduction to setting and using breakpoints	95
Reasons for using breakpoints	95
Briefly about setting breakpoints	96
Breakpoint types	96
Breakpoint icons	97
Breakpoints in the C-SPY simulator	98
Breakpoints in the C-SPY hardware drivers	98
Breakpoint consumers	99
Procedures for setting breakpoints	100
Various ways to set a breakpoint	101
Toggling a simple code breakpoint	101
Setting breakpoints using the dialog box	101
Setting a data breakpoint in the Memory window	103
Setting breakpoints using system macros	103
Useful breakpoint hints	104

Reference information on breakpoints	106
Breakpoints window	106
Breakpoint Usage dialog box	108
Code breakpoints dialog box	109
Log breakpoints dialog box	111
Data breakpoints dialog box	112
Immediate breakpoints dialog box	114
Complex breakpoints dialog box	115
Enter Location dialog box	118
Resolve Source Ambiguity dialog box	119
Monitoring memory and registers	121
Introduction to monitoring memory and registers	121
Briefly about monitoring memory and registers	121
C-SPY memory zones	122
Stack display	123
Reference information on memory and registers	124
Memory window	125
Memory Save dialog box	128
Memory Restore dialog box	129
Fill dialog box	130
Symbolic Memory window	131
Stack window	133
Register window	136
Collecting and using trace data	139
Introduction to using trace	139
Reasons for using trace	139
Briefly about trace	139
Requirements for using trace	140
Procedures for using trace	140
Getting started with trace in the C-SPY simulator	140
Trace data collection using breakpoints	141
Searching in trace data	141
Browsing through trace data	142

Reference information on trace	142
Trace window	143
Function Trace window	144
Timeline window	145
Trace Start breakpoints dialog box	148
Trace Stop breakpoints dialog box	149
Trace Expressions window	150
Find in Trace dialog box	151
Find in Trace window	152
Using the profiler	153
Introduction to the profiler	153
Reasons for using the profiler	153
Briefly about the profiler	153
Requirements for using the profiler	154
Procedures for using the profiler	155
Getting started using the profiler on function level	155
Getting started using the profiler on instruction level	155
Reference information on the profiler	156
Function Profiler window	157
Code coverage	161
Introduction to code coverage	161
Reasons for using code coverage	161
Briefly about code coverage	161
Requirements for using code coverage	161
Reference information on code coverage	162
Code Coverage window	162
Interrupts	165
Introduction to interrupts	165
Briefly about the interrupt simulation system	165
Interrupt characteristics	166
C-SPY system macros for interrupt simulation	167
Target-adapting the interrupt simulation system	167

Procedures for interrupts	168
Simulating a simple interrupt	168
Reference information on interrupts	169
Interrupts dialog box	170
Using C-SPY macros	173
Introduction to C-SPY macros	173
Reasons for using C-SPY macros	173
Briefly about using C-SPY macros	174
Briefly about setup macro functions and files	174
Briefly about the macro language	175
Procedures for using C-SPY macros	175
Registering C-SPY macros—an overview	176
Executing C-SPY macros—an overview	176
Using the Macro Configuration dialog box	177
Registering and executing using setup macros and setup files	178
Executing macros using Quick Watch	179
Executing a macro by connecting it to a breakpoint	180
Reference information on the macro language	181
Macro functions	181
Macro variables	182
Macro strings	182
Macro statements	183
Formatted output	184
Reference information on reserved setup macro function names	186
Reference information on C-SPY system macros	187
The C-SPY Command Line Utility—cspybat	219
Using C-SPY in batch mode	219
Invocation syntax	219
Output	220
Using an automatically generated batch file	220
Summary of C-SPY command line options	221
Reference information on C-SPY command line options ...	223

Debugger options	237
Setting debugger options	237
Reference information on debugger options	239
Setup	239
Images	241
Plugins	242
Reference information on C-SPY driver options	243
AVR ONE! 1	243
AVR ONE! 2	245
Communication	246
Extra Options	247
CCR	248
Serial Port	249
ICE200	251
JTAGICE 1	253
JTAGICE 2	255
JTAGICE3 1	256
JTAGICE3 2	258
JTAGICE mkII 1	259
JTAGICE mkII 2	262
Dragon 1	263
Dragon 2	265
Third-Party Driver options	266
Additional information on C-SPY drivers	269
Reference information on the C-SPY simulator	269
Simulator menu	270
Reference information on the C-SPY JTAGICE driver	270
JTAGICE menu	271
Reference information on the C-SPY JTAGICE mkII/Dragon driver	271
JTAGICE mkII menu	271
Dragon menu	272
Fuse Handler dialog box	273

Reference information on the C-SPY JTAGICE3 driver	275
JTAGICE3 menu	275
Fuse Handler dialog box	276
Reference information on the C-SPY AVR ONE! driver	278
AVR ONE! menu	278
Fuse Handler dialog box	279
Reference information on the C-SPY ICE200 driver	281
ICE200 menu	281
ICE200 Options dialog box	282
Reference information on the CCR driver	284
CCR menu	284
Resolving problems	285
Using C-SPY macros for transparent commands	285
Index	287

Tables

1: Typographic conventions used in this guide	25
2: Naming conventions used in this guide	25
3: Driver differences	33
4: C-SPY assembler symbols expressions	83
5: Handling name conflicts between hardware registers and assembler labels	83
6: Effects of display format setting on different types of expressions	89
7: Effects of display format setting on different types of expressions	91
8: Available breakpoints in C-SPY hardware drivers	98
9: C-SPY macros for breakpoints	104
10: C-SPY driver profiling support	154
11: Project options for enabling the profiler	155
12: Project options for enabling code coverage	162
13: Timer interrupt settings	169
14: Examples of C-SPY macro variables	182
15: C-SPY setup macros	186
16: Summary of system macros	187
17: __cancelInterrupt return values	189
18: __disableInterrupts return values	190
19: __driverType return values	191
20: __enableInterrupts return values	191
21: __evaluate return values	192
22: __isBatchMode return values	192
23: __loadImage return values	193
24: __openFile return values	197
25: __readFile return values	199
26: __setCodeBreak return values	202
27: __set Complex Break return values	205
28: __setDataBreak return values	207
29: __setLogBreak return values	208
30: __setSimBreak return values	209
31: __setTraceStartBreak return values	210

32: __setTraceStopBreak return values	212
33: __sourcePosition return values	212
34: __unloadImage return values	216
35: cspybat parameters	219
36: C-SPY command line options	221
37: Options specific to the C-SPY drivers you are using	237

Figures

1: C-SPY and target systems	31
2: C-SPY JTAGICE communication overview	36
3: C-SPY JTAGICE mkII communication overview	38
4: C-SPY JTAGICE3 communication overview	40
5: C-SPY AVR ONE! communication overview	42
6: C-SPY ICE200 communication overview	44
7: Finishing the ICE200 firmware upgrade	45
8: C-SPY CCR driver communication overview	46
9: Get Alternative File dialog box	51
10: Example applications	54
11: Debug menu	57
12: Images window	60
13: Images window context menu	61
14: Get Alternative File dialog box	61
15: C-SPY highlighting source location	67
16: C-SPY Disassembly window	69
17: Disassembly window context menu	71
18: Call Stack window	73
19: Call Stack window context menu	73
20: Terminal I/O window	74
21: Ctrl codes menu	75
22: Input Mode dialog box	75
23: Terminal I/O Log File dialog box	75
24: Debug Log window (message window)	76
25: Debug Log window context menu	76
26: Log File dialog box	77
27: Report Assert dialog box	78
28: Autostep settings dialog box	78
29: Viewing assembler variables in the Watch window	85
30: Auto window	86
31: Locals window	87

32: Watch window	87
33: Watch window context menu	88
34: Statics window	89
35: Statics window context menu	90
36: Quick Watch window	92
37: Symbols window	93
38: Symbols window context menu	93
39: Resolve Symbol Ambiguity dialog box	94
40: Breakpoint icons	97
41: Modifying breakpoints via the context menu	102
42: Breakpoints window	106
43: Breakpoints window context menu	107
44: Breakpoint Usage dialog box	108
45: Code breakpoints dialog box	109
46: Log breakpoints dialog box	111
47: Data breakpoints dialog box	112
48: Immediate breakpoints dialog box	114
49: Complex breakpoints dialog box	115
50: Enter Location dialog box	118
51: Resolve Source Ambiguity dialog box	119
52: Zones in C-SPY	122
53: Memory window	125
54: Memory window context menu	127
55: Memory Save dialog box	128
56: Memory Restore dialog box	129
57: Fill dialog box	130
58: Symbolic Memory window	131
59: Symbolic Memory window context menu	132
60: Stack window	133
61: Stack window context menu	135
62: Register window	136
63: The Trace window in the simulator	143
64: Function Trace window	144
65: Timeline window	145

66: Timeline window context menu for the Call Stack Graph	146
67: Trace Start breakpoints dialog box	148
68: Trace Stop breakpoints dialog box	149
69: Trace Expressions window	150
70: Find in Trace dialog box	151
71: Find in Trace window	152
72: Instruction count in Disassembly window	156
73: Function Profiler window	157
74: Function Profiler window context menu	159
75: Code Coverage window	162
76: Code coverage window context menu	164
77: Simulated interrupt configuration	166
78: Interrupt Setup dialog box	170
79: Macro Configuration dialog box	177
80: Quick Watch window	179
81: Debugger setup options	239
82: Debugger images options	241
83: Debugger plugin options	242
84: AVR ONE! 1 options	243
85: AVR ONE! 2 options	245
86: Communication options	246
87: C-SPY driver extra options	247
88: CCR options	248
89: Serial port options	249
90: ICE200 options	251
91: JTAGICE 1 options	253
92: JTAGICE 2 options	255
93: JTAGICE3 1 options	256
94: JTAGICE3 2 options	258
95: JTAGICE mkII 1 options	259
96: JTAGICE mkII 2 options	262
97: Dragon 1 options	263
98: Dragon 2 options	265
99: C-SPY Third-Party Driver options	266

100: Simulator menu	270
101: The JTAGICE menu	271
102: The JTAGICE mkII menu	271
103: The JTAGICE mkII menu when the debugger is running	272
104: The Dragon menu	272
105: The Dragon menu when the debugger is running	272
106: The JTAGICE mkII/Dragon Fuse Handler dialog box	273
107: The JTAGICE3 menu	275
108: The JTAGICE3 menu when the debugger is running	275
109: The JTAGICE3 Fuse Handler dialog box	276
110: The AVR ONE! menu	278
111: The AVR ONE! menu when the debugger is running	278
112: The AVR ONE! Fuse Handler dialog box	279
113: The ICE200 menu	281
114: The ICE200 Options dialog box	282
115: The CCR menu	284

Preface

Welcome to the *C-SPY® Debugging Guide for AVR*. 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 want to get the most out of the features available in C-SPY. In addition, you should have working knowledge of:

- The C or C++ programming language
- Application development for embedded systems
- The architecture and instruction set of the AVR microcontroller (refer to the chip manufacturer's documentation)
- 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 23.

How to use this guide

If you are new to using IAR Embedded Workbench, we suggest that you first read the guide *Getting Started with IAR Embedded Workbench®* for an overview of the tools and the features that the IDE offers.

If you already have had some experience using IAR Embedded Workbench, but need refreshing on how to work with the IAR Systems development tools, the tutorials which you can find in the IAR Information Center is a good place to begin. The process of managing projects and building, as well as editing, is described in the *IDE Project Management and Building Guide*, whereas information about how to use C-SPY for debugging is described in this guide.

This guide describes a number of *topics*, where each topic section contains an introduction which also covers concepts related to the topic. This will give you a good understanding of the features in C-SPY. Furthermore, the topic section provides procedures with step-by-step descriptions to help you use the features. Finally, each topic section gives all relevant reference information.

We also recommend the Glossary which you can find in the *IDE Project Management and Building Guide* if you should encounter any unfamiliar terms in the IAR Systems user and reference guides.

What this guide contains

This is a brief outline and summary of the chapters in this guide:

- *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.
- *Working with 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.
- *Using breakpoints* describes the breakpoint system and the various ways to set breakpoints.
- *Monitoring memory and registers* shows how you can examine memory and registers.
- *Collecting and using trace data* describes how you can inspect the program flow up to a specific state using trace data.
- *Using the profiler* 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.
- *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.
- *Using 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.

- *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. For information about:

- System requirements and information about how to install and register the IAR Systems products, refer to 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, see the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, see the *IDE Project Management and Building Guide*.
- Programming for the IAR C/C++ Compiler for AVR, see the *IAR C/C++ Compiler Reference Guide for AVR*.
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, see the *IAR Linker and Library Tools Reference Guide*.
- Programming for the IAR Assembler for AVR, see the *AVR IAR Assembler Reference Guide*.
- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.
- Using the IAR CLIB Library, see the *IAR C Library Functions Reference Guide*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for AVR, see the *AVR® IAR Embedded Workbench® IDE Migration Guide*.
- Developing safety-critical applications using the MISRA C guidelines, see the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Comprehensive 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**, contains information and news about the AVR microcontrollers.
- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, contains information about the Embedded C++ standard.

Document conventions

When, in this text, 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 6.5\avr\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:





Style	Used for
computer	<ul style="list-style-type: none"> Source code examples and file paths. Text on the command line. Binary, hexadecimal, and octal numbers.
<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 command.
[a b c]	An optional part of a command with alternatives.
{a b c}	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> A cross-reference within this guide or to another guide. Emphasis.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the 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® referred to in this guide:

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

Table 2: Naming conventions used in this guide

Brand name	Generic term
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)

The IAR C-SPY Debugger

This chapter introduces you to the IAR 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. More specifically, this means:

- Introduction to C-SPY
- Debugger concepts
- C-SPY drivers overview
- The IAR C-SPY Simulator
- The C-SPY JTAGICE driver
- The C-SPY JTAGICE mkII/Dragon driver
- The C-SPY JTAGICE3 driver
- The C-SPY AVR ONE! driver
- The C-SPY ICE200 driver
- The C-SPY Crypto Controller ROM-monitor driver.

Introduction to C-SPY

This section covers these topics:

- 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. Any variable and expression can be evaluated in one-shot views. You can easily both monitor and log values of a defined set of expressions during a longer period of time. You have instant

control over local variables, and real-time data is displayed non-intrusively. Finally, the last referred variables are displayed automatically.

- 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
- UBROF, Intel-extended, and Motorola input formats supported
- Optional terminal I/O emulation.

RTOS AWARENESS

C-SPY supports real-time OS aware debugging. These operating systems are currently supported:

- Micrium μ C/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.

Provided that one or more real-time operating system plugin modules are supported for the IAR Embedded Workbench version you are using, you can load one for use with C-SPY. A C-SPY RTOS awareness plugin module gives you a high level of control and visibility over an application built on top of a real-time operating system. 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 each chapter of this part of the documentation. The IAR Systems user documentation uses the terms described in this section when referring to these concepts.

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:

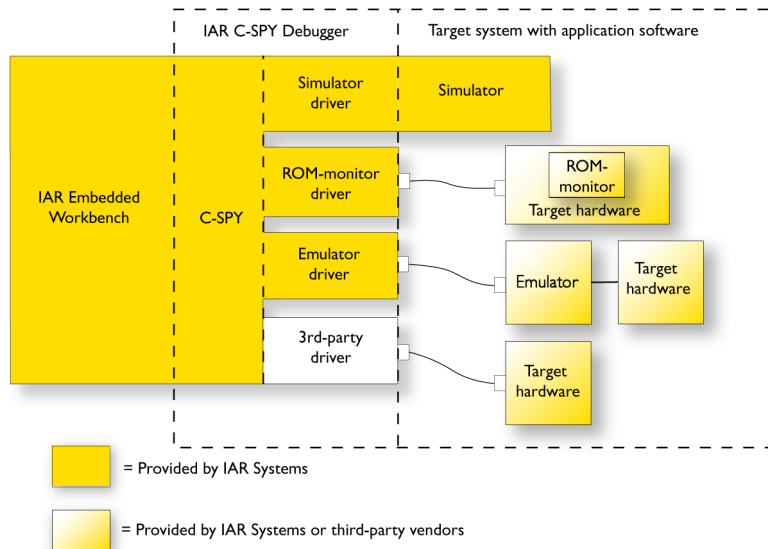


Figure 1: C-SPY and target systems

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 33.

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, refer to 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® JTAGICE
- AVR® JTAGICE mkII/AVR® Dragon
- AVR® JTAGICE3
- AVR® AVR ONE!
- 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 the IAR Embedded Workbench, you can also load debugger drivers supplied by a third-party vendor; see *Third-Party Driver options*, page 266.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

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

Feature	Simulator	JTAGICE					
		JTAGICE	mkII/ Dragon	JTAGICE3	AVR ONE!	ICE200	CCR
Code breakpoints	Unlimited	4 ¹	4 ¹	Unlimited ⁴	Unlimited ⁴	Unlimited	Unlimited
Data breakpoints	x	2 ¹	2 ¹	2	2	--	--
Execution in real time	--	x	x	x	x	x	x
Zero memory footprint	x	x	x	x	x	x	x
Simulated interrupts	x	--	--	--	--	--	--
Real interrupts	--	x	x	x	x	x	x
Cycle counter	x	--	--	--	--	x	--

Table 3: Driver differences

Feature	Simulator	JTAGICE					
		JTAGICE	mkII/ Dragon	JTAGICE3	AVR ONE!	ICE200	CCR
Code coverage	x	--	--	--	--	--	--
Data coverage	x	--	--	--	--	--	--
Function/instruction profiler	x	--	x ²	x ^{2, 3}	x ^{2, 3}	x	x ²
Trace	x	--	--	--	--	--	--

Table 3: Driver differences (Continued)

1 The sum of code and data breakpoints can never exceed 4—the number of available hardware breakpoints. For more information, see *Breakpoints in the C-SPY hardware drivers*, page 98.

2 Cycle counter statistics are not available.

3 Software breakpoints must be enabled.

4 As many code breakpoints as the device provides and that are available. For information about the number of available code breakpoints for your device, see the device-specific documentation provided by Atmel® Corporation.

For more information about breakpoints, see *Using breakpoints*, page 95.

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.

FEATURES

In addition to the general features in C-SPY, the simulator also provides:

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

SELECTING THE SIMULATOR DRIVER

Before starting C-SPY, you must choose the simulator driver:

- 1 In the IDE, choose **Project>Options** and click the **Setup** tab in the **Debugger** category.
- 2 Choose **Simulator** from the **Driver** drop-down list.

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.

FEATURES

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

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via the serial port.

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.

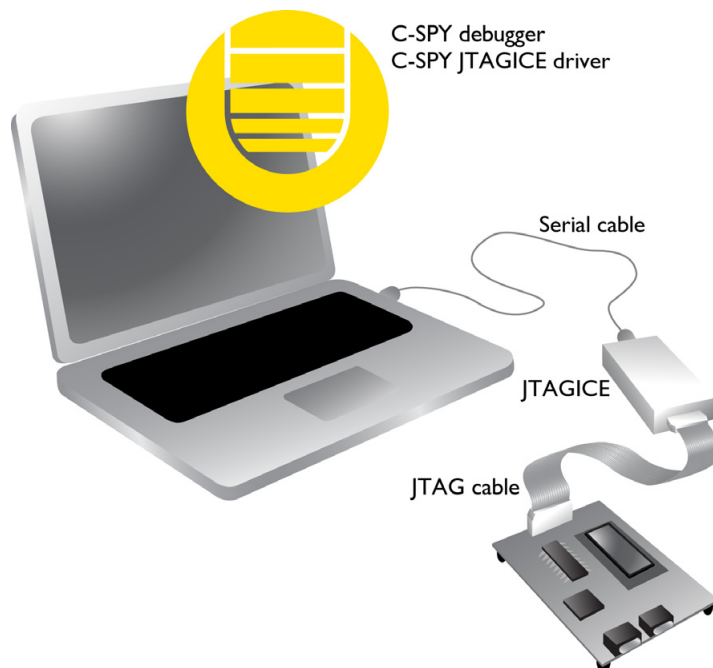


Figure 2: C-SPY JTAGICE communication overview

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 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 debugging 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 JTAGICE firmware should be version 0x73 or higher to ensure compatibility with C-SPY. The latest firmware can be obtained in the AVR Studio package available for download from Atmel Corporation's web site.

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 or the USB port to communicate with Atmel AVR JTAGICE mkII. JTAGICE mkII communicates with the JTAG, the PDI, or the debugWIRE interface on the microcontroller.

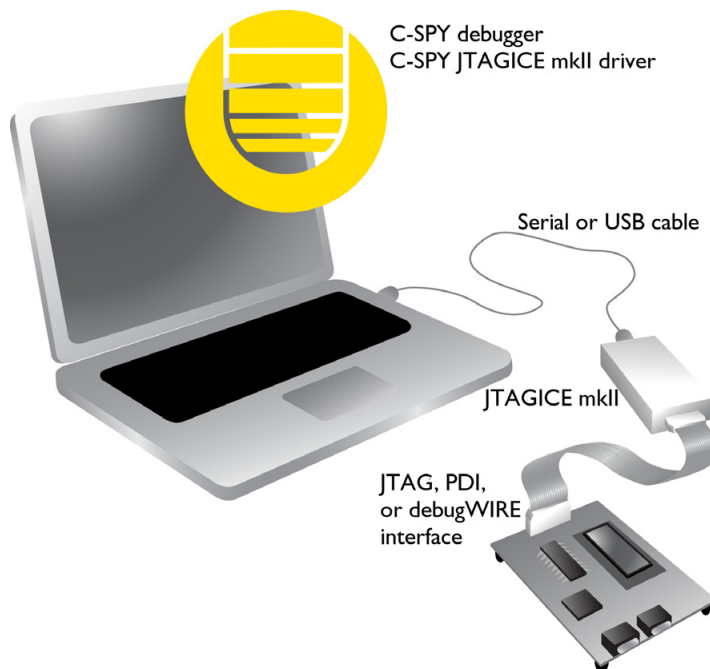


Figure 3: C-SPY JTAGICE mkII communication overview

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 debugging 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 273.

C-SPY automatically detects if the firmware needs to be updated to ensure compatibility with C-SPY. The latest firmware can be obtained in the AVR Studio package available for download from Atmel Corporation's web site.

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, DW, or ISP—on the microcontroller.

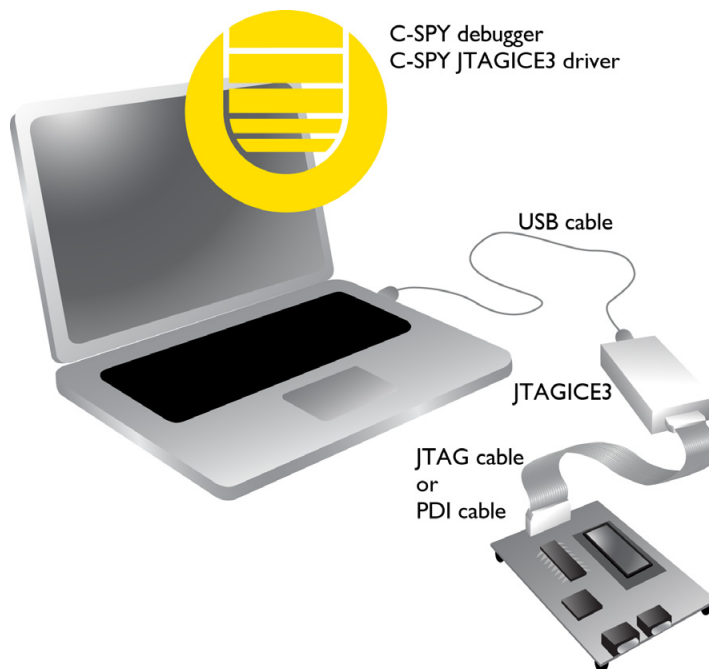


Figure 4: C-SPY JTAGICE3 communication overview

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 debugging 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 273.

C-SPY automatically detects if the firmware needs to be updated to ensure compatibility with C-SPY. The latest firmware can be obtained in the AVR Studio package available for download from Atmel Corporation's web site.

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, DW, or ISP—on the microcontroller.

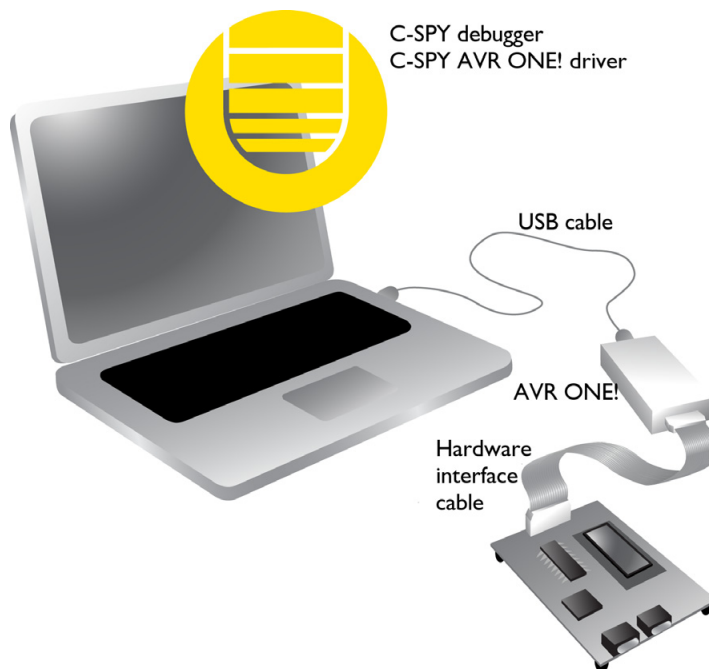


Figure 5: C-SPY AVR ONE! communication overview

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 debugging 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 279.

C-SPY automatically detects if the firmware needs to be updated to ensure compatibility with C-SPY. The latest firmware can be obtained in the AVR Studio package available for download from Atmel Corporation's web site.

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.

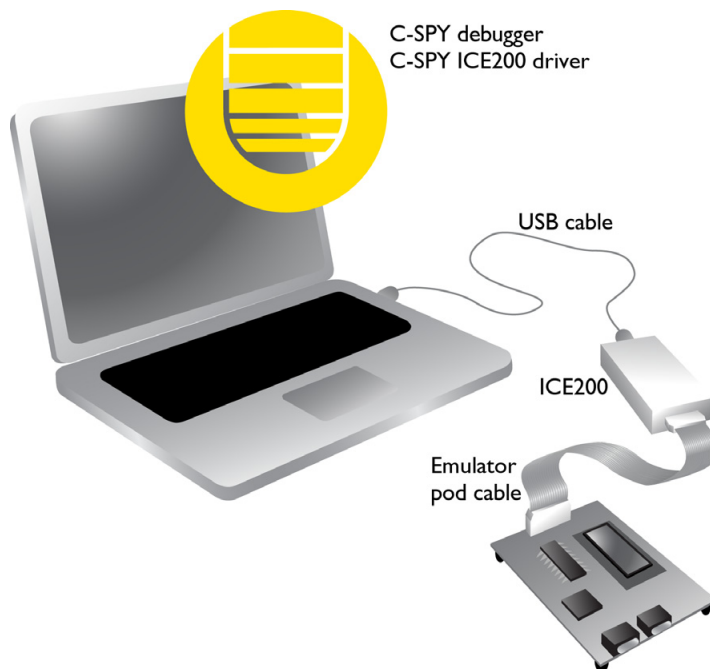


Figure 6: C-SPY ICE200 communication overview

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.

UPGRADING THE FIRMWARE

When the C-SPY ICE200 driver starts up, it will check that the ICE200 firmware version is compatible. If an old firmware version is detected, it must be upgraded. A dialog box appears, which informs you about the required upgrade.

To upgrade the firmware version, follow this procedure:

- 1 First, turn OFF the power to ICE200. Then turn the power ON. A warning message will appear if you omit turning off the power supply.
- 2 You will then be asked to locate the upgrade program—`ICE200upgrade.exe`.

The upgrade program is distributed with AVR Studio. The program can also be started separately without C-SPY.

- 3 When the firmware upgrade has been fully completed, reset the hardware and then click OK in this dialog box.



Figure 7: Finishing the ICE200 firmware upgrade

You can now start your debug session.

The C-SPY Crypto Controller ROM-monitor driver

The C-SPY Crypto Controller ROM-monitor (CCR) driver is available as a separate product and is installed as an add-on to the IAR Embedded Workbench for AVR. 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 RS-232 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.

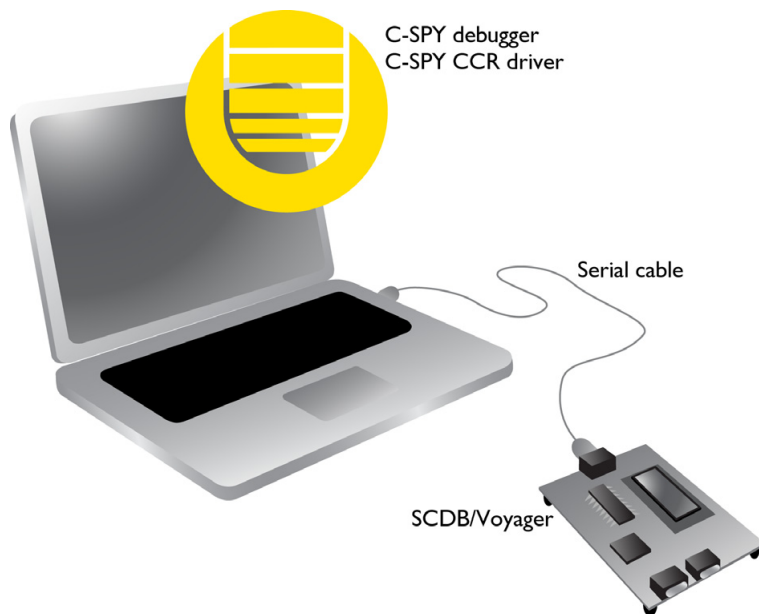


Figure 8: C-SPY CCR driver communication overview

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

This chapter helps you get started using C-SPY®. More specifically, this means:

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

Setting up C-SPY

This section describes the steps involved for setting up C-SPY.

More specifically, you will get information about:

- 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.

Note: You can only choose a driver you have installed on your computer.

- 2 In the **Category** list, select the appropriate C-SPY driver and make your settings.

For information about these options, see *Debugger options*, page 237.

- 3 Click **OK**.

4 Choose **Tools>Options>Debugger** to configure:

- The debugger behavior
- 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 52.

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 the debugger 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. Note that if you have selected the option **Use UBROF reset vector**, the program counter will contain the address of the label configured as 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 not limited.

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 *Briefly about setup macro functions and files*, page 174. For an example of how to use a setup macro file, see the chapter *Initializing target hardware before C-SPY starts*, page 53.

To register a setup macro file:

- I 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 52.

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.

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 242.

Starting C-SPY

When you have set up the debugger, you are ready to start a debug session; this section describes the steps involved.

More specifically, you will get information about:

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

STARTING THE DEBUGGER

You can choose to start the debugger with or without loading the current project.



To start C-SPY and load the current project, click the **Download and Debug** button. Alternatively, choose **Project>Download and Debug**.



To start C-SPY without reloading the current project, 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:

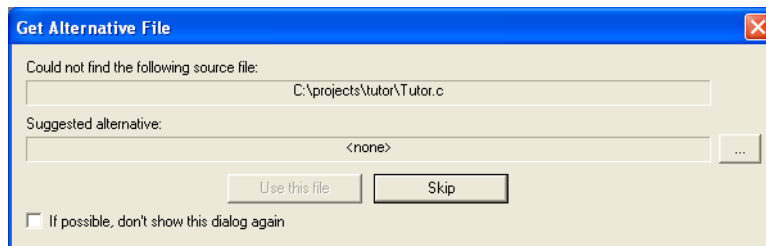


Figure 9: Get Alternative File dialog box

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 61.

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 241.
- 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 *Procedures for using C-SPY macros*, page 175.

To display a list of loaded images:

Choose **Images** from the **View** menu. The Images window is displayed, see *Images window*, page 60.

Adapting for target hardware

This section provides information about how to describe the target hardware to C-SPY, and how you can make C-SPY initialize the target hardware before your application is downloaded to memory.

More specifically, you will get information about:

- 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 49. They contain device-specific information such as:

- Memory information for device-specific memory zones, see *C-SPY memory zones*, page 122
- 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 165.

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. The syntax is

described in the files. Note, however, that the format of these descriptions might be updated in future upgrade versions 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.

For information about how to load a device description file, see *Selecting a device description file*, page 49.

INITIALIZING TARGET HARDWARE BEFORE C-SPY STARTS

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. For example:

- 1 Create a new text file and define your macro function. For example, a macro that enables external SDRAM might look like this:

```
/* Your macro function. */
enableExternalSDRAM()
{
    __message "Enabling external SDRAM\n";
    __writeMemory32( /* Place your code here. */ );
    /* And more code here, if needed. */
}

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

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

- 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

IAR Embedded Workbench comes with example applications. You can use these examples to get started using the development tools from IAR Systems or simply to

verify that contact has been established with your target board. 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.

RUNNING AN EXAMPLE PROJECT

To run an example project:

- 1 Choose **Help>Information Center** and click **EXAMPLE PROJECTS**.
- 2 Browse to the example that matches the specific evaluation board or starter kit you are using.

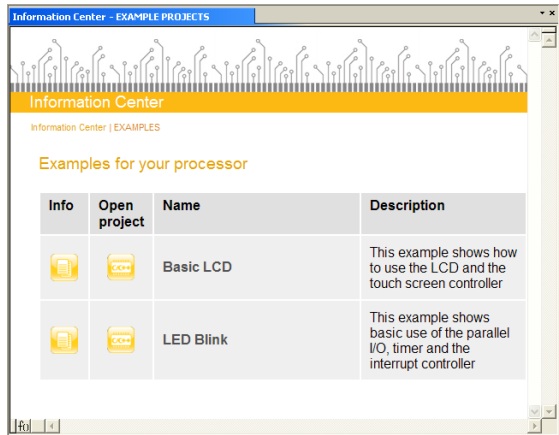


Figure 10: Example applications

Click the **Open Project** button.

- 3 In the dialog box that appears, choose a destination folder for your project location. Click **Select** to confirm your choice.
- 4 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.
- 5 To view the project settings, select the project and choose **Options** from the context menu. Verify the settings for **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 237.

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



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

7 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 285.



8 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

This section gives reference information about these windows and dialog boxes:

- *C-SPY Debugger main window*, page 55
- *Images window*, page 60
- *Get Alternative File dialog box*, page 61

See also:

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

C-SPY Debugger main window

When you start the debugger, 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 when C-SPY is running:

Debug	Provides commands for executing and debugging the source application, see <i>Debug menu</i> , page 57. Most of the commands are also available as icon buttons on the debug toolbar.
Simulator	Provides access to the dialog boxes for setting up interrupt simulation and memory access checking. This menu is only available when the C-SPY Simulator is used, see <i>Simulator menu</i> , page 270.
JTAGICE	Provides commands specific to the C-SPY JTAGICE driver. This menu is only available when the driver is used, see <i>JTAGICE menu</i> , page 271.
JTAGICE mkII	Provides commands specific to the C-SPY JTAGICE mkII driver. This menu is only available when the driver is used, see <i>JTAGICE mkII menu</i> , page 271.
Dragon	Provides commands specific to the C-SPY Dragon driver. This menu is only available when the driver is used, see <i>Dragon menu</i> , page 272.
JTAGICE3	Provides commands specific to the C-SPY JTAGICE3 driver. This menu is only available when the driver is used, see <i>JTAGICE3 menu</i> , page 275.
AVR ONE!	Provides commands specific to the C-SPY AVR ONE! driver. This menu is only available when the driver is used, see <i>AVR ONE! menu</i> , page 278.
ICE200	Provides commands specific to the C-SPY ICE200 driver. This menu is only available when the driver is used, see <i>ICE200 menu</i> , page 281.
CCR	Provides commands specific to the C-SPY CCR driver. This menu is only available when the driver is used, see <i>CCR menu</i> , page 284.

Debug menu

The **Debug** menu is available when C-SPY is running. 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.

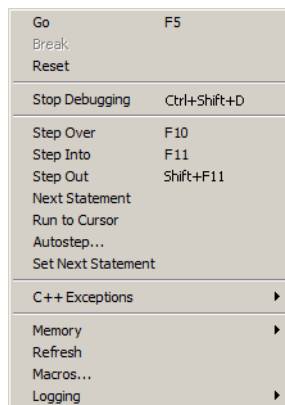


Figure 11: Debug menu

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.



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 autosteppping, see *Autostep settings dialog box*, page 78.

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 128.

Memory>Restore

Displays a dialog box where you can load the contents of a file in Intel-extended or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 129.

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 the Macro Configuration dialog box*, page 177.

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 77.

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 75.

C-SPY windows

Depending on the C-SPY driver you are using, these windows specific to C-SPY are available when C-SPY is running:

- 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:

Enter	Makes an item editable and saves the new value.
Esc	Cancels a new value.

In windows where you can edit the **Expression** field, 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
```

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
```

Images window

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

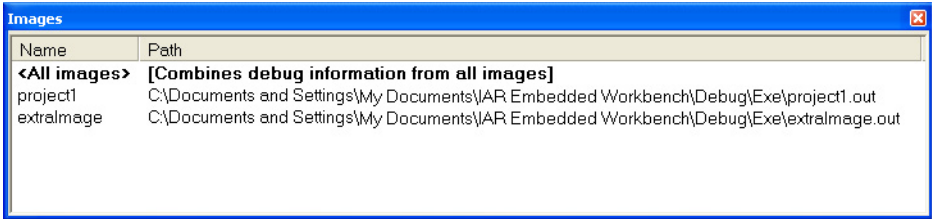


Figure 12: Images window

The Images 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.

Display area

This area lists the loaded images in these columns:

Name	The name of the loaded image.
Path	The path to the loaded image.

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.

Context menu

This context menu is available:

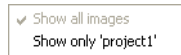


Figure 13: Images window context menu

These commands are available:

Show all images	Shows debug information for all loaded debug images.
Show only <i>image</i>	Shows debug information for the selected debug image.

Related information

For related information, see:

- *Loading multiple images*, page 51
- *Images*, page 241
- *__loadImage*, page 193.

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.

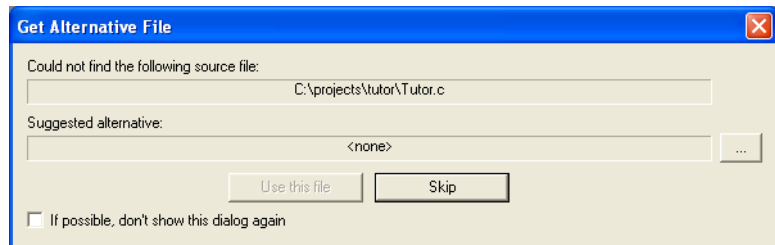


Figure 14: Get Alternative File dialog box

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 50.

Executing your application

This chapter contains information about executing your application in C-SPY®. More specifically, this means:

- Introduction to application execution
- Reference information on application execution.

Introduction to application execution

This section covers these topics:

- Briefly about application execution
- Source and disassembly mode debugging
- Single stepping
- 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 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 78.

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.

RUNNING THE APPLICATION



Go

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



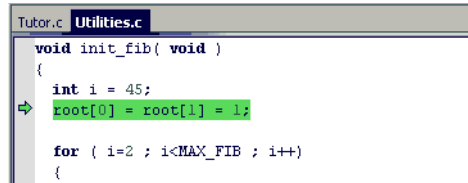
Run to Cursor

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++)
    {

```

Figure 15: C-SPY highlighting source location

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 74 and *Terminal I/O Log File dialog box*, page 75.

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

This section gives reference information about these windows and dialog boxes:

- *Disassembly window*, page 69
- *Call Stack window*, page 73
- *Terminal I/O window*, page 74
- *Terminal I/O Log File dialog box*, page 75
- *Debug Log window*, page 76
- *Log File dialog box*, page 77
- *Report Assert dialog box*, page 78

- Autostep settings dialog box, page 78.
- See also Terminal I/O options in *IDE Project Management and Building Guide*.

Disassembly window

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

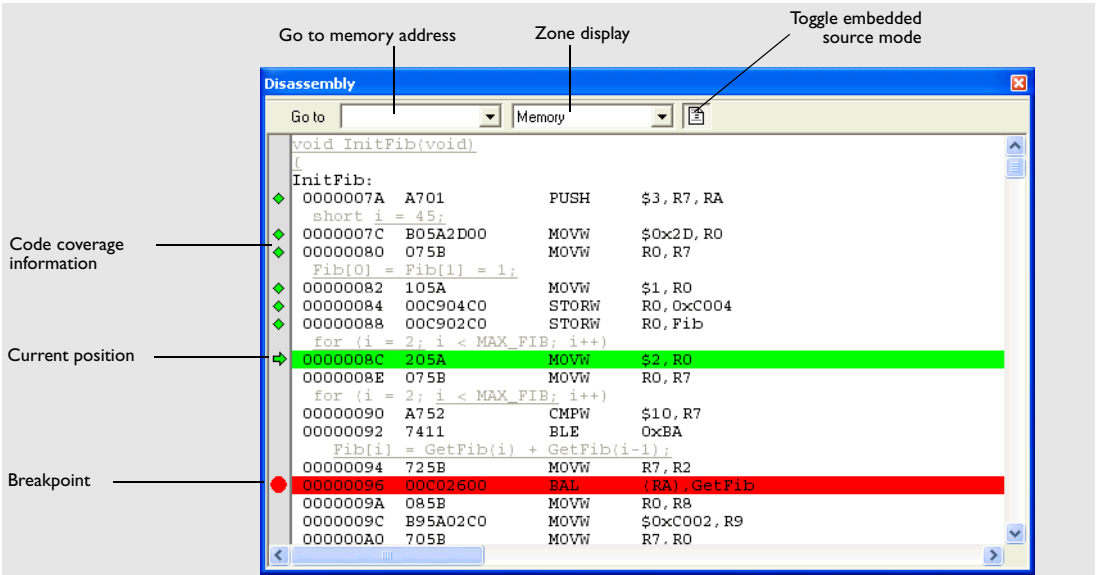


Figure 16: C-SPY Disassembly window

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.

Toolbar

The toolbar contains:

Go to	The location you want to view. This can be a memory address, or the name of a variable, function, or label.
Zone display	Lists the available memory zones to display, see <i>C-SPY memory zones</i> , page 122.
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 Disassembly 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 Call Stack window or between items in the Trace 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>Using breakpoints</i> , page 95.
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:

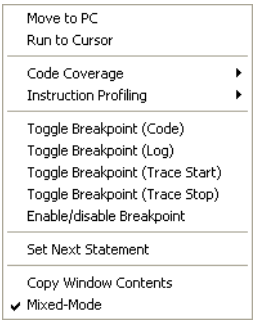


Figure 17: Disassembly window context menu

Note: The contents of this menu are dynamic, which means it might look different depending 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	<p>Displays a submenu that provides commands for controlling code coverage. This command is only enabled if the driver you are using supports it.</p> <p>Enable, toggles code coverage on or off.</p> <p>Show, toggles the display of code coverage on or off.</p> <p>Executed code is indicated by a green diamond.</p> <p>Clear, clears all code coverage information.</p>
Instruction Profiling	<p>Displays a submenu that provides commands for controlling instruction profiling. This command is only enabled if the driver you are using supports it.</p> <p>Enable, toggles instruction profiling on or off.</p> <p>Show, toggles the display of instruction profiling on or off.</p> <p>For each instruction, the left-side margin displays how many times the instruction has been executed.</p> <p>Clear, clears all instruction profiling information.</p>

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 <i>Code breakpoints dialog box</i> , page 109.
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 <i>Log breakpoints dialog box</i> , page 111.
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 <i>Trace Start breakpoints dialog box</i> , page 148.
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 <i>Trace Stop breakpoints dialog box</i> , page 149.
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.

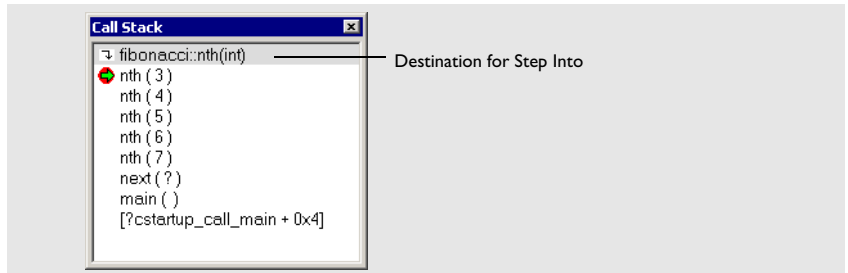


Figure 18: Call Stack window

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.

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 value of the parameters, or empty if the function does not take any parameters.

Context menu

This context menu is available:

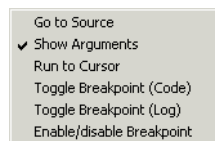


Figure 19: Call Stack window context menu

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.
Enable/Disable Breakpoint	Enables or disables the selected breakpoint.

Terminal I/O window

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

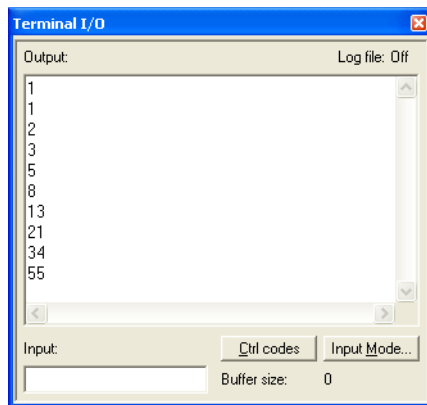


Figure 20: Terminal I/O window

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.

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.

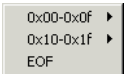


Figure 21: Ctrl codes menu

Input Mode

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

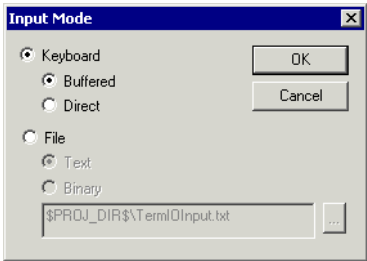


Figure 22: Input Mode dialog box

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**.

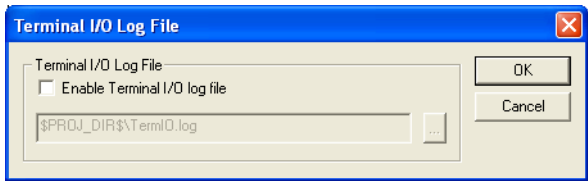


Figure 23: Terminal I/O Log File dialog box

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

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**.

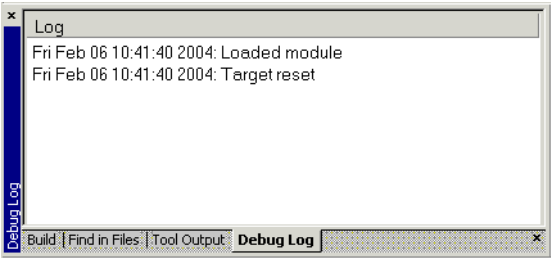


Figure 24: Debug Log window (message window)

This window displays debugger output, such as diagnostic messages, macro-generated output, event log messages, and information about trace. This output is only available when C-SPY is running. 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>
```

Context menu

This context menu is available:

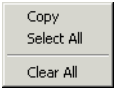


Figure 25: Debug Log window context menu

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**.

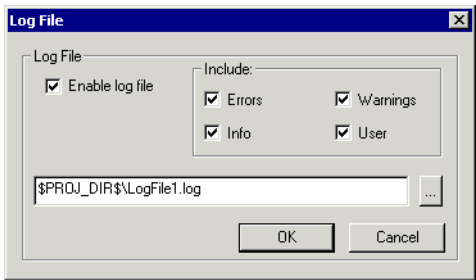


Figure 26: Log File dialog box

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

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. 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 <code>__message</code> statement.

Use the browse button, to override the default file and location of the log file (the default filename extension is `log`).

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.

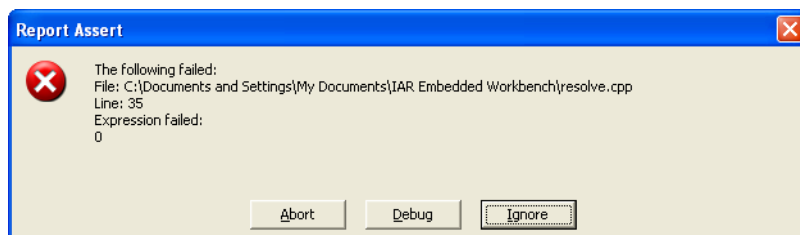


Figure 27: Report Assert dialog box

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.

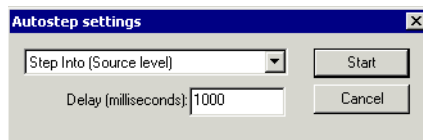


Figure 28: Autostep settings dialog box

Use this dialog box to customize autostepping.

The drop-down menu lists the available step commands.

Delay

Specify the delay between each step in milliseconds.

Working with variables and expressions

This chapter describes how variables and expressions can be used in C-SPY®. More specifically, this means:

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

Introduction to working with variables and expressions

This section covers these topics:

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

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 Quick Watch window gives 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 Trace-related windows let you inspect the program flow up to a specific state. For more information, see *Collecting and using trace data*, page 139.

Details about using these windows

All the windows are easy to use. You can add, modify, and remove expressions, and change the display format.

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 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 and the Quick Watch window where it is not relevant.

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
#asm_label
#R2
#PC
my_macro_func(19)
```

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.

Assembler symbols

Assembler symbols can be assembler labels or register names. That is, general purpose registers, such as R0–R31, and special purpose registers, such as the program counter and the status register. 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 52.

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

Example	What it does
#pc++	Increments the value of the program counter.
myptr = #label7	Sets myptr to the integral address of label7 within its zone.

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
#pc	Refers to the program counter.
#`pc`	Refers to the assembler label pc.

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 136.

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 175.

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 181.

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**.

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:

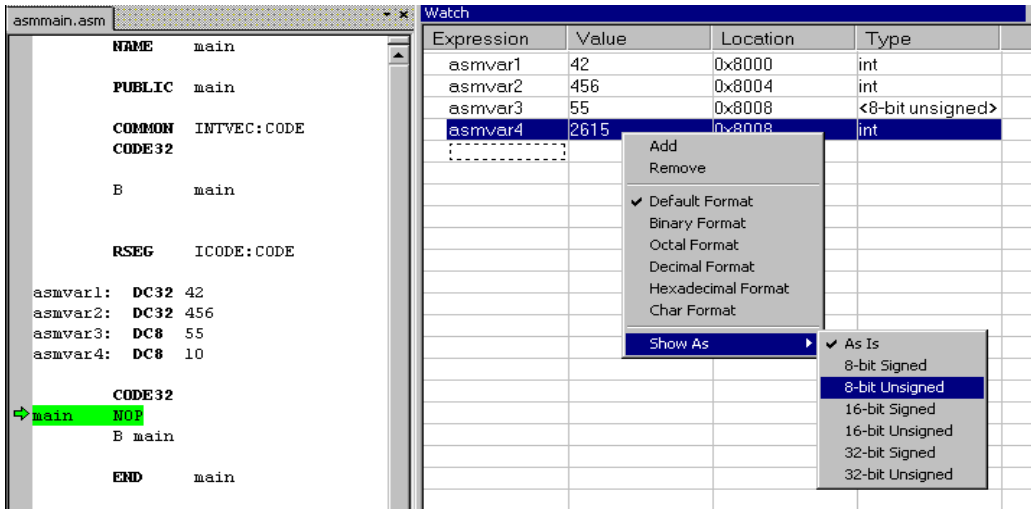


Figure 29: Viewing assembler variables in the Watch window

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.

Reference information on working with variables and expressions

This section gives reference information about these windows and dialog boxes:

- *Auto window*, page 86
- *Locals window*, page 87
- *Watch window*, page 87
- *Statics window*, page 89
- *Quick Watch window*, page 92
- *Symbols window*, page 93
- *Resolve Symbol Ambiguity dialog box*, page 94.

For trace-related reference information, see *Reference information on trace*, page 142.

Auto window

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

Expression	Value	Location	Type
root[0]	1	DATA:0x000062	unsigned int
⊕ root	<array>	DATA:0x000062	unsigned int[10]
root[1]	1	DATA:0x000064	unsigned int
i	6	R25:R24	short

Figure 30: Auto window

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.

Context menu

For more information about the context menu, see *Watch window*, page 87.

Locals window

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

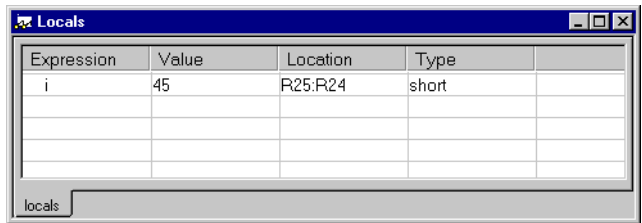


Figure 31: Locals window

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

Context menu

For more information about the context menu, see *Watch window*, page 87.

Watch window

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

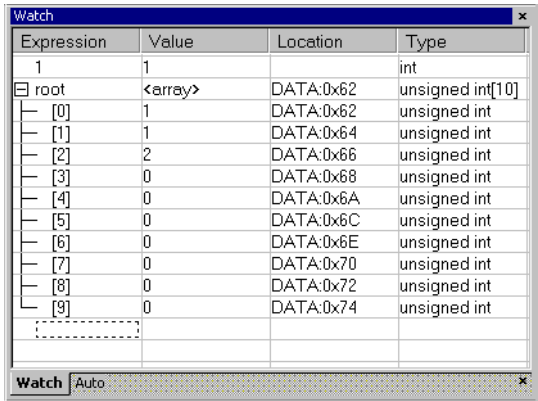


Figure 32: Watch window

Use this window to monitor the values of C-SPY expressions or variables. 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.

Context menu

This context menu is available:

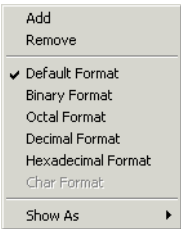


Figure 33: Watch window context menu

These commands are available:

Add	Adds an expression.
Remove	Removes the selected expression.
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, see Table 6, <i>Effects of display format setting on different types of expressions</i> . Your selection of display format is saved between debug sessions.
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 <i>Viewing assembler variables</i> , page 85.

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

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

Table 6: Effects of display format setting on different types of expressions

Statics window

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

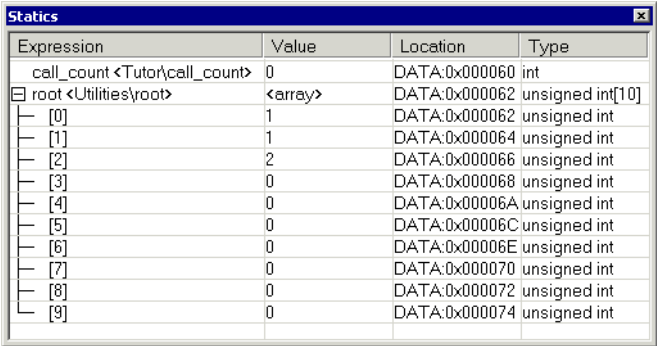


Figure 34: Statics window

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.

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 be displayed, or choose **Select All** or **Deselect All** from the context menu.

- 3 When you have made your selections, choose **Select statics** from the context menu to toggle back to the normal display mode.

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	<p>The value of the variable. Values that have changed are highlighted in red.</p> <p>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.</p>
Location	The location in memory where this variable is stored.
Type	The data type of the variable.

Context menu

This context menu is available:

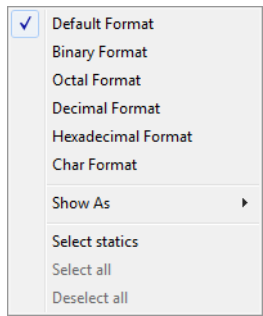


Figure 35: Statics window context menu

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, see Table 6, <i>Effects of display format setting on different types of expressions</i> . Your selection of display format is saved between debug sessions. These commands are available if a selected line in the Statics window contains a variable.
Select Statics	Lists all variables with static storage duration. Select the variables you want to be monitored. When you have made your selections, select this menu command again to toggle back to normal display mode.
Select all	Selects all variables.
Deselect all	Deselects all variables.

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

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

Table 7: Effects of display format setting on different types of expressions

Quick Watch window

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

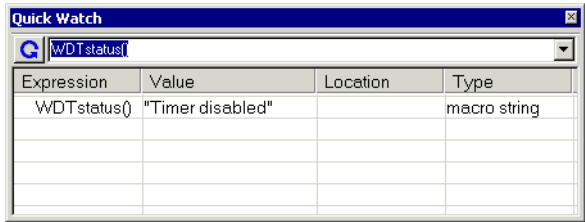


Figure 36: Quick Watch window

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.

To evaluate an expression:

- 1 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:

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



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

For an example, see *Executing macros using Quick Watch*, page 179.

Context menu

For more information about the context menu, see *Watch window*, page 87.

In addition, the menu contains the **Add to Watch window** command, which adds the selected expression to the Watch window.

Symbols window

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

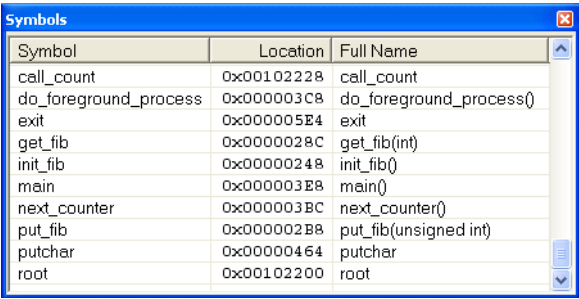


Figure 37: Symbols window

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.

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:

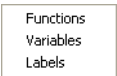


Figure 38: Symbols window context menu

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.

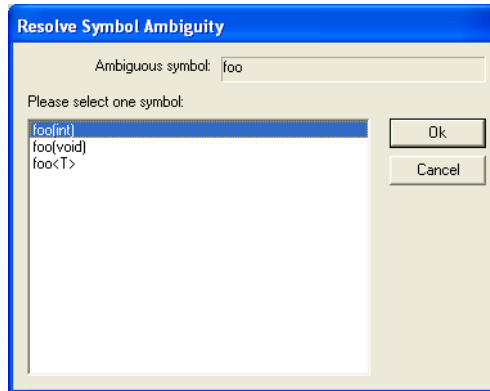


Figure 39: Resolve Symbol Ambiguity dialog box

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.

Using breakpoints

This chapter describes breakpoints and the various ways to define and monitor them. More specifically, this means:

- Introduction to setting and using breakpoints
- Procedures for setting breakpoints
- Reference information on breakpoints.

Introduction to setting and using breakpoints

This section introduces 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 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** dialog box also lists all internally used breakpoints, see *Breakpoint consumers*, page 99.

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 64.

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 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.

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 AVR ONE! driver supports complex breakpoints. Complex breakpoints use the functionality of the AVR ONE! firmware and are faster than data breakpoints and code breakpoints.

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:

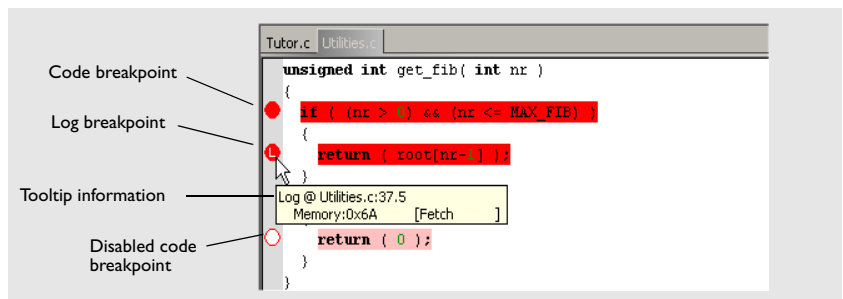


Figure 40: Breakpoint icons

Note: In the figure, *Memory* is an example memory zone.



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** dialog box.

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, except for complex breakpoints, and you can set an unlimited amount of breakpoints.

BREAKPOINTS IN THE C-SPY HARDWARE 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:

C-SPY hardware driver	Code breakpoints	Data breakpoints
JTAGICE		
using hardware breakpoints ⁸	4 ¹	2 ¹
using software breakpoints	Unlimited	Unlimited
JTAGICE mkII		
using hardware breakpoints ^{3, 8}	4 ^{1, 2}	2 ^{1, 4}
using software breakpoints	Unlimited	Unlimited
JTAGICE3		
using hardware breakpoints ^{3, 5, 7, 8}	4 ^{1, 2}	2 ⁶
using software breakpoints	Unlimited	Unlimited
AVR ONE!		
using hardware breakpoints ^{3, 5, 7, 8}	4 ^{1, 2}	2 ⁶
using software breakpoints	Unlimited	Unlimited
ICE200	Unlimited	None

Table 8: Available breakpoints in C-SPY hardware drivers

C-SPY hardware driver	Code breakpoints	Data breakpoints
CCR	Unlimited	None

Table 8: Available breakpoints in C-SPY hardware drivers (Continued)

1 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.

2 If software breakpoints are enabled, the number of code breakpoints are unlimited.

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

4 Data breakpoints are not available when the debugWIRE interface is used.

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

6 If complex breakpoints are used, data breakpoints are not available, and vice versa.

7 Note that a complex breakpoint uses all available hardware breakpoints.

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

For JTAGICE mkII, 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.

When software breakpoints 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** dialog box 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 when the debugger system is running. This means that they are not visible in the Breakpoints window.
- The linker option **With I/O emulation modules** has been selected.

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

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:

AVR ONE!	<i>System breakpoints on, page 246</i>
JTAGICE	<i>System breakpoints on, page 255</i>
JTAGICE3	<i>System breakpoints on, page 259</i>
JTAGICE mkII	<i>System breakpoints on, page 262</i>
Dragon	<i>System breakpoints on, page 265</i>

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

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.

Procedures for setting breakpoints

This section gives you step-by-step descriptions about how to set and use breakpoints.

More specifically, you will get information about:

- 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:

- Using the **Toggle Breakpoint** command toggles a code breakpoint. This command is available both from the **Tools** menu and from the context menus in the editor window and in the Disassembly window.
- Right-clicking in the left-side margin of the editor window or the Disassembly window toggles a 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

Toggling 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:

- Double-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.

To set a new breakpoint:

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

- 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.

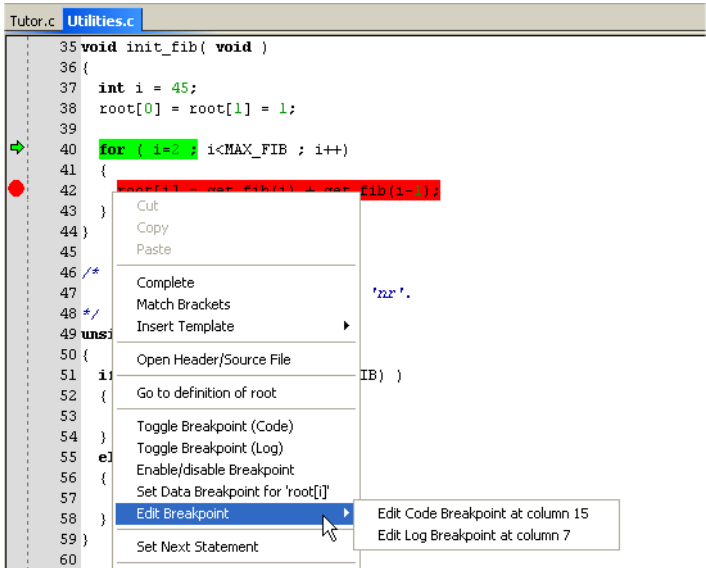


Figure 41: Modifying breakpoints via the context menu

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	Simulator	JTAGICE	JTAGICE mkII	JTAGICE3	AVR ONE!	ICE200	CCR
__setCodeBreak	x	x	x	x	x	x	x
__setComplexBreak	—	—	—	x	x	—	—
__setDataBreak	x	—	—	x	x	—	—
__setLogBreak	x	x	x	x	x	x	x
__setSimBreak	x	—	—	—	—	—	—
__setTraceStartBreak	x	—	—	—	—	—	—
__setTraceStopBreak	x	—	—	—	—	—	—
__clearBreak	x	x	x	x	x	x	x

Table 9: C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 187.

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 *Registering and executing using setup macros and setup files*, page 178.

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

This section gives reference information about these windows and dialog boxes:

- *Breakpoints window*, page 106
- *Breakpoint Usage dialog box*, page 108
- *Code breakpoints dialog box*, page 109
- *Log breakpoints dialog box*, page 111
- *Data breakpoints dialog box*, page 112
- *Immediate breakpoints dialog box*, page 114
- *Complex breakpoints dialog box*, page 115
- *Enter Location dialog box*, page 118
- *Resolve Source Ambiguity dialog box*, page 119.

See also:

- *Reference information on C-SPY system macros*, page 187
- *Reference information on trace*, page 142.

Breakpoints window

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

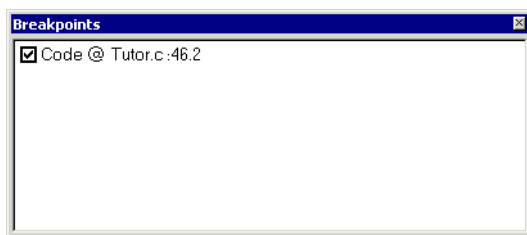


Figure 42: Breakpoints window

The Breakpoints 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.

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:

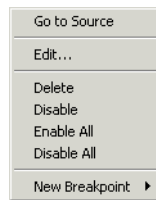


Figure 43: Breakpoints window context menu

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 dialog box

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

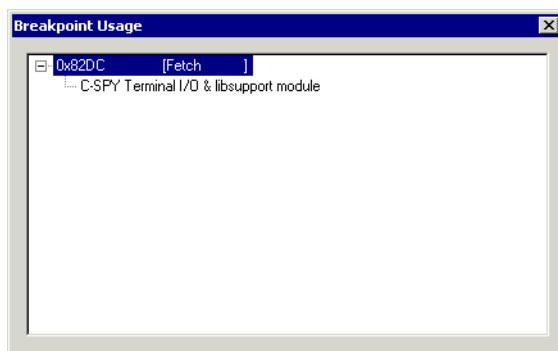


Figure 44: Breakpoint Usage dialog box

The **Breakpoint Usage** dialog box 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 dialog box depends on the C-SPY driver you are using.

The dialog box 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. If your target system has a limited number of hardware breakpoints and software breakpoints are not enabled, exceeding the number of available hardware breakpoints will cause the debugger to single step. This will significantly reduce the execution speed. Therefore, in a debugger system with a limited amount of hardware breakpoints, you can use the **Breakpoint Usage** dialog box 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.

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.

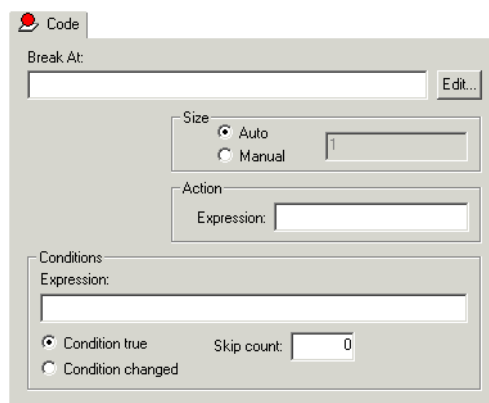


Figure 45: Code breakpoints dialog box

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

Note: The **Code** breakpoints dialog box depends on the C-SPY driver you are using. This figure reflects the C-SPY simulator. For information about support for breakpoints in the C-SPY driver you are using, see *Breakpoints in the C-SPY hardware drivers*, page 98.

Break At

Specify the 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 118.

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 an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.
Conditions		Specify simple or complex conditions:
	Expression	Specify a valid expression conforming to the C-SPY expression syntax.
	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.

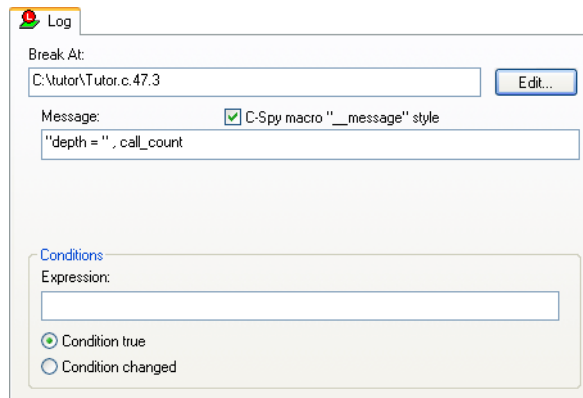


Figure 46: Log breakpoints dialog box

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

Note: The **Log** breakpoints dialog box depends on the C-SPY driver you are using. This figure reflects the C-SPY simulator. For information about support for breakpoints in the C-SPY driver you are using, see *Breakpoints in the C-SPY hardware drivers*, page 98.

Break At

Specify the location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 118.

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 184.

Conditions

Specify simple or complex conditions:

Expression	Specify a valid expression conforming to the C-SPY expression syntax.
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.

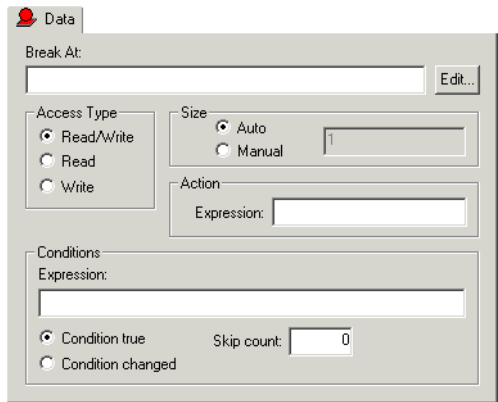


Figure 47: Data breakpoints dialog box

This dialog box is available for the C-SPY JTAGICE driver and the JTAGICE mkII driver, unless the debugWIRE interface is used.

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.

Note: The **Data** breakpoints dialog box depends on the C-SPY driver you are using. This figure reflects the C-SPY simulator. For information about support for breakpoints in the C-SPY driver you are using, see *Breakpoints in the C-SPY hardware drivers*, page 98.

Break At

Specify the location for the breakpoint in the **Break At** text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 118.

Access Type

Selects the type of memory access that triggers data breakpoints:

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. 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. Select between two different ways 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.

Action

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

Conditions

Specify simple or complex conditions:

Expression	Specify a valid expression conforming to the C-SPY expression syntax.
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 a break occurs (integer).

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.

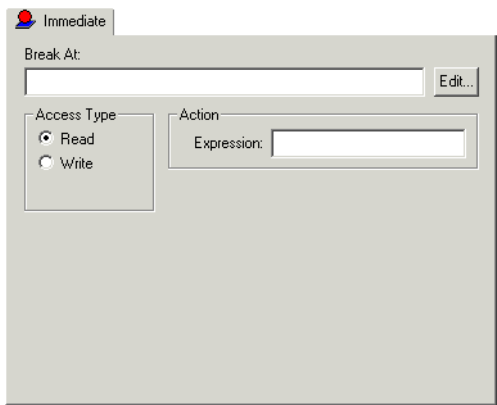


Figure 48: Immediate breakpoints dialog box

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.

Break At

Specify the location for the breakpoint in the **Break At** text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 118.

Access Type

Selects the type of memory access that triggers immediate breakpoints:

Read	Reads from location.
Write	Writes to location.

Action

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

Complex breakpoints dialog box

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

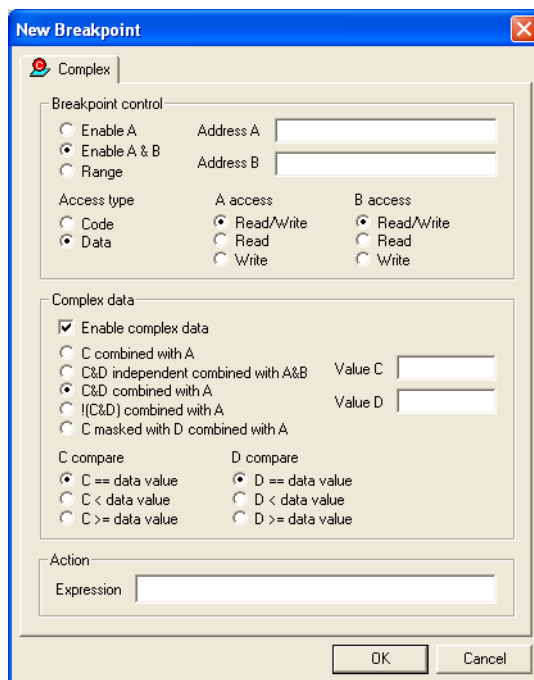


Figure 49: Complex breakpoints dialog box

This dialog box is available for the C-SPY AVR ONE! driver.

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 AVR ONE! firmware and are faster than data breakpoints and code breakpoints.

Note: A complex breakpoint uses all available hardware breakpoints.

Breakpoint control

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

Enable A	Breaks at the address specified in the Address A text box.
Enable A&B	Breaks both at the address specified in Address A and at the address in Address B .
Range	Breaks when an address from Address A up to and including Address 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 Range , only A access is available.
A access/B access	Specifies the type of memory access that triggers complex breakpoints. Read/Write , Reads from or writes to location Read , Reads from location Write , 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.

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

**C&D independent
combined with A&B**

Breaks when **Address A** is accessed using **A access** and **Value C** matches the memory contents at **Address A** according to **C compare**

or

when **Address B** is accessed using **B access** and **Value D** matches the memory contents at **Address B** according to **D compare**.

C&D combined with A

Breaks when **Address A** is accessed using **A access** *and* **Value C** matches the memory contents at **Address A** according to **C compare** *and* **Value D** matches the memory contents at **Address A** according to **D compare**.

(C&D) combined with A

Breaks when **Address A** is accessed using **A access** *and* **Value C** does not match the memory contents of **Address A** according to **C compare** *and/or* **Value D** does not match the memory contents of **Address A** according to **D compare**.

**C masked with D
combined with A**

Breaks when **Address A** is accessed using **A access** *and* **Value C** masked with **Value D** matches the memory contents at **Address A** according to **C compare**.

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.

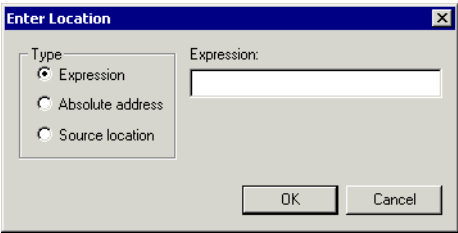


Figure 50: Enter Location dialog box

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:

Expression	Any expression that evaluates to a valid address, such as a function or variable name. Code breakpoints are set on functions and data breakpoints are set on variable names. For example, <code>my_var</code> refers to the location of the variable <code>my_var</code> , and <code>arr[3]</code> refers to the third element of the array <code>arr</code> .
Absolute address	An absolute location on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code> . Zone refers to C-SPY memory zones and specifies in which memory the address belongs. For example <code>Memory: 0x42</code> .

Source location

A location in the C source code using the syntax:

`{file_path}.row.column`.

file_path 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:\my_projects\Utilities.c}.22.3` sets a breakpoint on the third character position on line 22 in the source file `Utilities.c`.

Note that the Source location type is usually meaningful only for code breakpoints.

Resolve Source Ambiguity dialog box

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

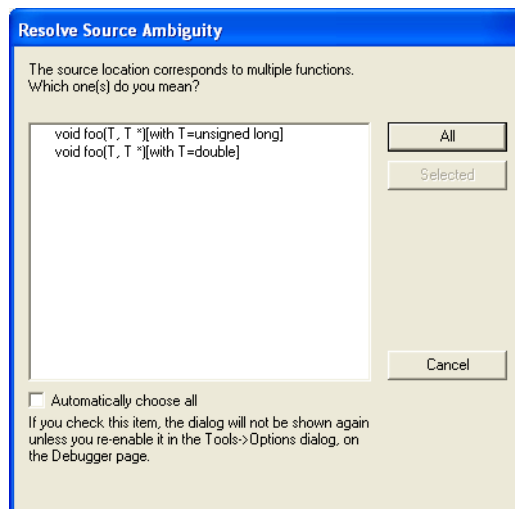


Figure 51: Resolve Source Ambiguity dialog box

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*.

Monitoring memory and registers

This chapter describes how to use the features available in C-SPY® for examining memory and registers. More specifically, this means information about:

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

Introduction to monitoring memory and registers

This section covers these topics:

- 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. Due to 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 in the debugger cover the whole AVR memory range.

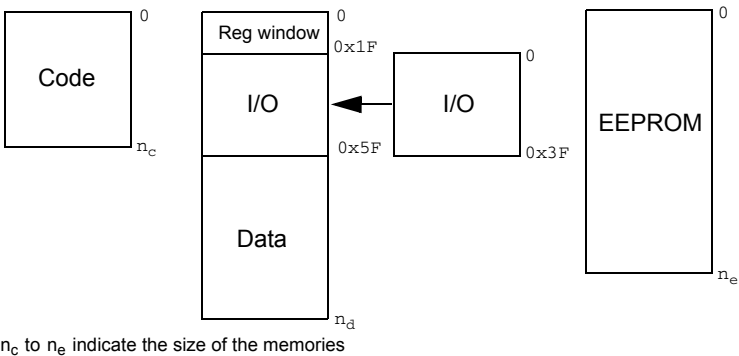


Figure 52: Zones in C-SPY

These zones are available: CODE, DATA, EEPROM, and IO_SPACE.

Memory zones are used in several contexts, most importantly in the Memory and Disassembly windows. Use the **Zone** box in these windows to choose which memory zone to display.

Device-specific zones

Memory information for device-specific zones are defined in the *device description files*. By default, there are a number of address zones in the debugger. If you load a device description file, additional zones that adhere to the specific memory layout are defined.

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 49 and *Modifying a device description file*, page 52.

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*.

Reference information on memory and registers

This section gives reference information about these windows and dialog boxes:

- *Memory window*, page 125
- *Memory Save dialog box*, page 128
- *Memory Restore dialog box*, page 129
- *Fill dialog box*, page 130
- *Symbolic Memory window*, page 131
- *Stack window*, page 133
- *Register window*, page 136.

Memory window

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

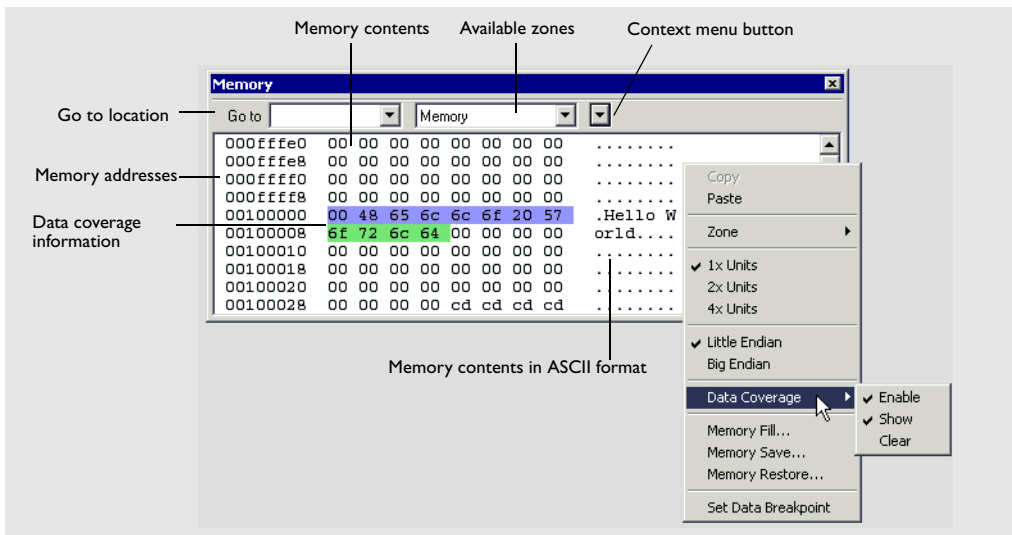


Figure 53: Memory window

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.

Toolbar

The toolbar contains:

Go to	The location you want to view. This can be a memory address, or the name of a variable, function, or label.
Zone display	Selects a memory zone to display, see <i>C-SPY memory zones</i> , page 122.
Context menu button	Displays the context menu, see <i>Context menu</i> , page 127.

Update Now	Updates the content of the Memory window while your application is executing. This button is only enabled if the C-SPY driver you are using has access to the target system memory while your application is executing.
Live Update	Updates the contents of the Memory window regularly while your application is executing. This button is only enabled if the C-SPY driver you are using has access to the target system memory while your application is executing. To set the update frequency, specify an appropriate frequency in the IDE Options>Debugger dialog box.

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:



Figure 54: Memory window context menu

These commands are available:

Copy, Paste	Standard editing commands.
Zone	Selects a memory zone to display, see <i>C-SPY memory zones</i> , page 122.
1x Units	Displays the memory contents in units of 8 bits.
2x Units	Displays the memory contents in units of 16 bits.
4x Units	Displays the memory contents in units of 32 bits.
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 <i>IDE Project Management and Building Guide</i> .
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 <i>IDE Project Management and Building Guide</i> .
Memory Fill	Displays a dialog box, where you can fill a specified area with a value, see <i>Fill dialog box</i> , page 130.
Memory Save	Displays a dialog box, where you can save the contents of a specified memory area to a file, see <i>Memory Save dialog box</i> , page 128.
Memory Restore	Displays a dialog box, where you can load the contents of a file in Intex-hex or Motorola s-record format to a specified memory zone, see <i>Memory Restore dialog box</i> , page 129.
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 <i>Setting a data breakpoint in the Memory window</i> , page 103.

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.

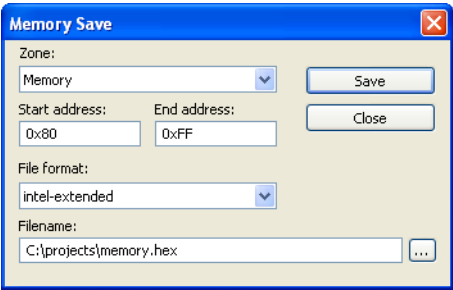


Figure 55: Memory Save dialog box

Use this dialog box to save the contents of a specified memory area to a file.

Zone

Selects a memory zone.

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.

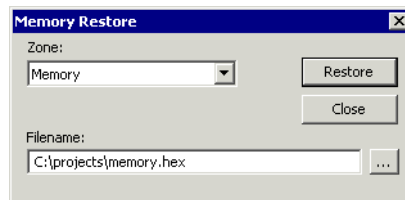


Figure 56: Memory Restore dialog box

Use this dialog box to load the contents of a file in Intel-extended or Motorola S-record format to a specified memory zone.

Zone

Selects a memory zone.

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.

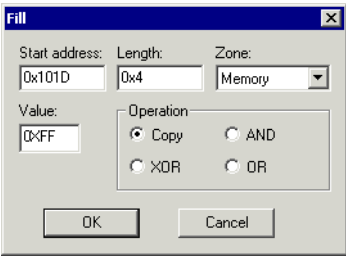


Figure 57: Fill dialog box

Use this dialog box to fill a specified area of memory with a value.

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.

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 when the debugger is running.

Symbolic Memory					
Go to		DATA	Previous	Next	
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				

Figure 58: Symbolic Memory window

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.

Toolbar

The toolbar contains:

- Go to

The memory location or symbol you want to view.

Zone display	Selects a memory zone to display, see <i>C-SPY memory zones</i> , page 122.
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:

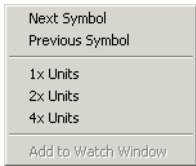


Figure 59: Symbolic Memory window context menu

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 in units of 8 bits. This applies only to rows which do not contain a variable.
2x Units	Displays the memory contents in units of 16 bits.
4x Units	Displays the memory contents in units of 32 bits.
Add to Watch Window	Adds the selected symbol to the Watch window.

Stack window

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

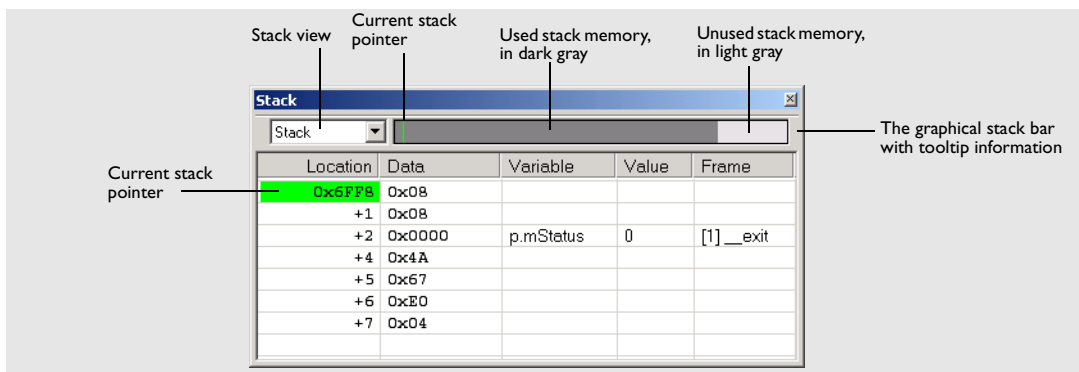


Figure 60: Stack window

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 99.

For information about options specific to the Stack window, see the *IDE Project Management and Building Guide*.

Toolbar

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:

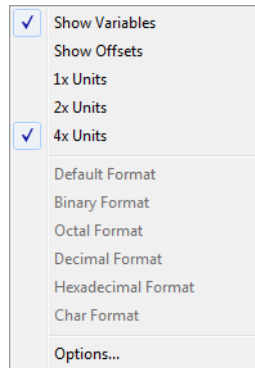


Figure 61: Stack window context menu

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 data in the Data column as single bytes.
2x Units	Displays data in the Data column as 2-byte groups.
4x Units	Displays data in the Data column 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, see Table 7, <i>Effects of display format setting on different types of expressions</i> . Your selection of display format is saved between debug sessions. These commands are available if a selected line in the Stack window contains a variable.
Options	Opens the IDE Options dialog box where you can set options specific to the Stack window, see the <i>IDE Project Management and Building Guide</i> .

Toolbar

CPU Registers

Selects which register group to display, by default CPU Registers. Additional register groups are predefined in the device description files that make all 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. To edit the contents of a register, click it, and modify the 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**.

Collecting and using trace data

This chapter gives you information about collecting and using trace data in C-SPY®. More specifically, this means:

- Introduction to using trace
- Procedures for using trace
- Reference information on trace.

Introduction to using trace

This section introduces trace.

These topics are covered:

- Reasons for using trace
- Briefly about trace
- Requirements for using trace.

See also *Using the profiler*, page 153.

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

Your target system must be able to 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 executed instructions 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. In the C-SPY simulator, you can also use the Trace Expressions window. Depending on your C-SPY driver, you can set various types of trace breakpoints to control the collection of trace data.

In addition, several other features in C-SPY also use trace data, features such as the Profiler, 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 target systems.

Procedures for using trace

This section gives you step-by-step descriptions about how to collect and use trace data.

More specifically, you will get information about:

- Getting started with trace in the C-SPY simulator
- Trace data collection using breakpoints
- Searching in trace data
- Browsing through trace data
- Reference information on trace.

GETTING STARTED WITH TRACE IN THE C-SPY SIMULATOR

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, choose **Simulator>Trace** to open the Trace window, and click the **Activate** button to enable collecting trace data.
- 2 Start the execution. When the execution stops, for instance because a breakpoint is triggered, trace data is displayed in the Trace window. For more information about the window, see *Trace window*, page 143.

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 148 and *Trace Stop breakpoints dialog box*, page 149, 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 In 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 different options, see *Find in Trace dialog box*, page 151.

- 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 152.

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

This section gives reference information about these windows and dialog boxes:

- *Trace window*, page 143
- *Function Trace window*, page 144
- *Timeline window*, page 145
- *Trace Start breakpoints dialog box*, page 148
- *Trace Stop breakpoints dialog box*, page 149
- *Trace Expressions window*, page 150
- *Find in Trace dialog box*, page 151
- *Find in Trace window*, page 152.

Trace window

The Trace window is available from the C-SPY driver menu.

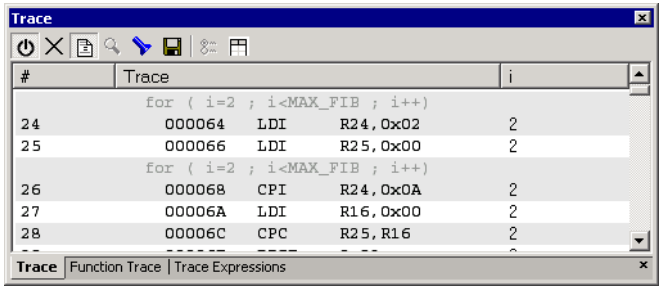









Figure 63: The Trace window in the simulator

This window displays the collected sequence of executed machine instructions. In addition, the window can display trace data for expressions.

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 <i>Browsing through trace data</i> , page 142. |
|  | Find | Displays a dialog box where you can perform a search, see <i>Find in Trace dialog box</i> , page 151. |
|  | 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 150.

Display area

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 <i>after</i> 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 <i>Trace Expressions window</i> , page 150.

Function Trace window

The Function Trace window is available from the C-SPY driver menu during a debug session.

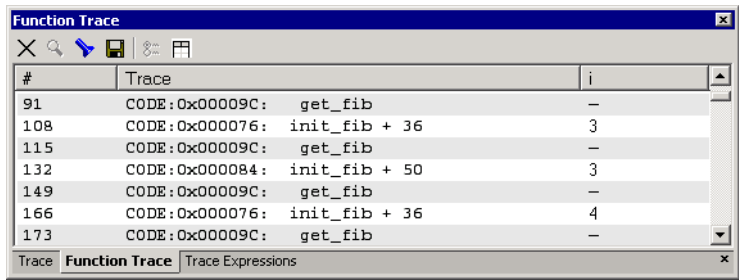


Figure 64: Function Trace window

This window is available for the C-SPY simulator.

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.

Toolbar

For information about the toolbar, see *Trace toolbar*, page 143.

Display area

For information about the columns in the display area, see *Display area*, page 144.

Timeline window

The Timeline window is available from the *C-SPY driver* menu during a debug session.

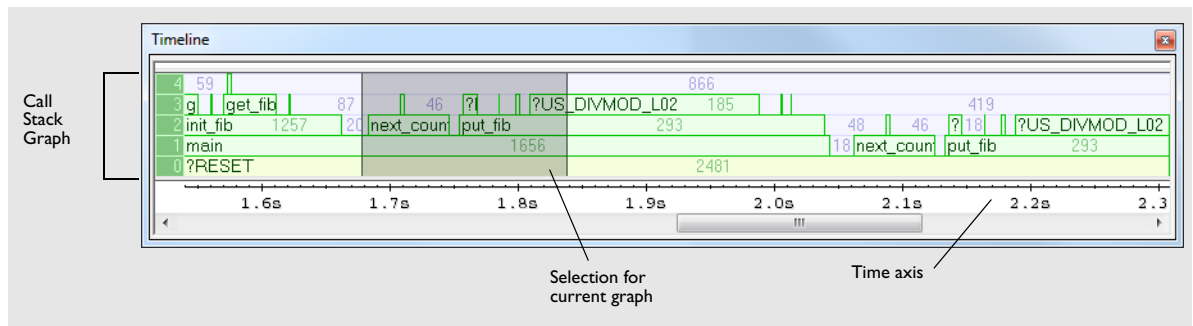


Figure 65: Timeline window

This window is available for the C-SPY simulator.

This window displays trace data—for the call stack—as a graph in relation to a time axis.

Display area

The display area contains the Call Stack graph.

At the bottom of the window, there is a time axis that uses seconds as the time unit.

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.

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:

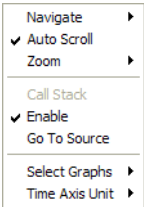


Figure 66: Timeline window context menu for the Call Stack Graph

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 recent collected data is automatically displayed.
Zoom	All graphs	<p>Commands for zooming the window, in other words, changing the time scale; choose between:</p> <p>Zoom to Selection makes the current selection fit the window. Shortcut key: Return.</p> <p>Zoom In zooms in on the time scale. Shortcut key: +.</p> <p>Zoom Out zooms out on the time scale. Shortcut key: -.</p> <p>10ns, 100ns, 1us, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.</p> <p>1ms, 10ms, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window.</p> <p>10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.</p>
Call Stack	Call Stack Graph	A heading that shows that the Call stack-specific commands below are available.
Interrupt	Interrupt Log Graph	A heading that shows that the Interrupt Log-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, <i>no data</i> 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	Common	Enables profiling time intervals in the Function Profiler window. Note that this command is only available if the C-SPY driver supports PC Sampling.
--------------------------	--------	--

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.

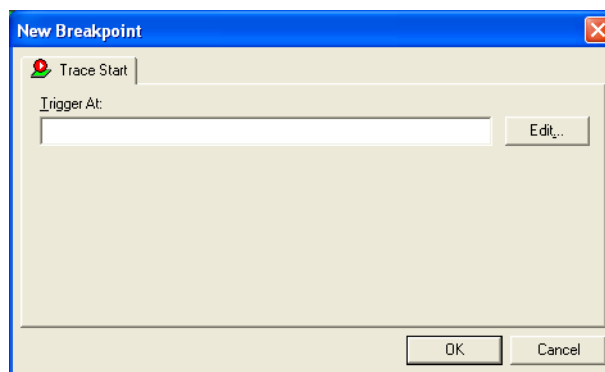


Figure 67: Trace Start breakpoints dialog box

This dialog box is available for the C-SPY simulator.

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.

Trigger At

Specify the location for the breakpoint in the text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 118.

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.

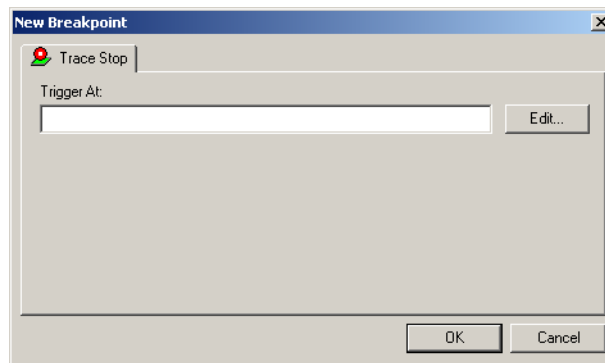


Figure 68: Trace Stop breakpoints dialog box

This dialog box is available for the C-SPY simulator.

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.

Trigger At

Specify the location for the breakpoint in the text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 118.

Trace Expressions window

The Trace Expressions window is available from the Trace window toolbar.

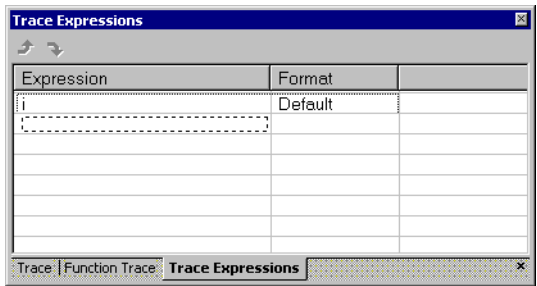


Figure 69: Trace Expressions window

This dialog box is available for the C-SPY simulator.

Use this window to specify, for example, a specific variable (or an expression) for which you want to collect trace data.

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.

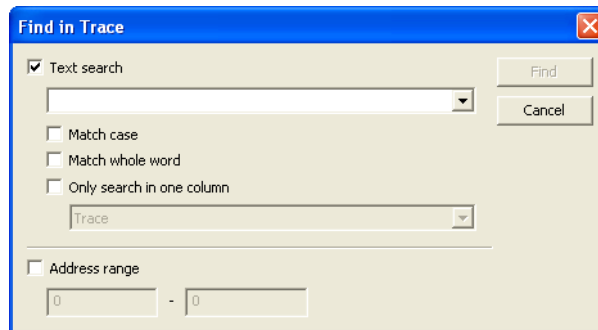


Figure 70: Find in Trace dialog box

This dialog box is available for the C-SPY simulator.

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 152.

See also *Searching in trace data*, page 141.

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.

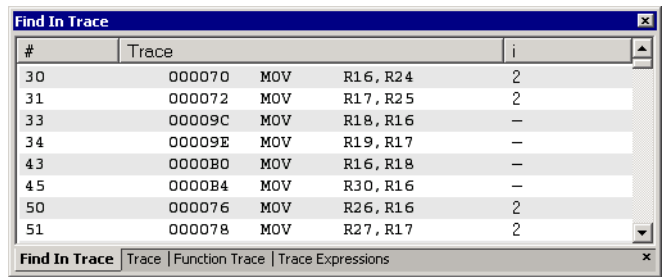


Figure 71: Find in Trace window

This dialog box is available for the C-SPY simulator.

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 151.

For more information, see *Searching in trace data*, page 141.

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.

Using the profiler

This chapter describes how to use the profiler in C-SPY®. More specifically, this means:

- Introduction to the profiler
- Procedures for using the profiler
- Reference information on the profiler.

Introduction to the profiler

This section introduces 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 the memory which uses the most efficient addressing mode. For detailed information about efficient memory usage, see the *IAR C/C++ Compiler Reference Guide for AVR*.

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 hardware 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 supports the profiler, and there are no specific requirements for using the profiler.

This table lists the C-SPY driver profiling support:

C-SPY driver	Trace (calls)	Trace (flat)
C-SPY simulator	X	X
JTAGICE	--	--
JTAGICE mkII	--	--
JTAGICE3	--	--
AVR ONE!	--	--
ICE200	--	--
CCR	--	--

Table 10: C-SPY driver profiling support

Procedures for using the profiler

This section gives you step-by-step descriptions about how to use the profiler.

More specifically, you will get information about:

- Getting started using the profiler on function level
- 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:

- 1 Make sure you build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Format>Debug information for C-SPY

Table 11: Project options for enabling the profiler

- 2 When you have built your application and started C-SPY, choose **Function Profiler** from the C-SPY driver menu 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.

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.

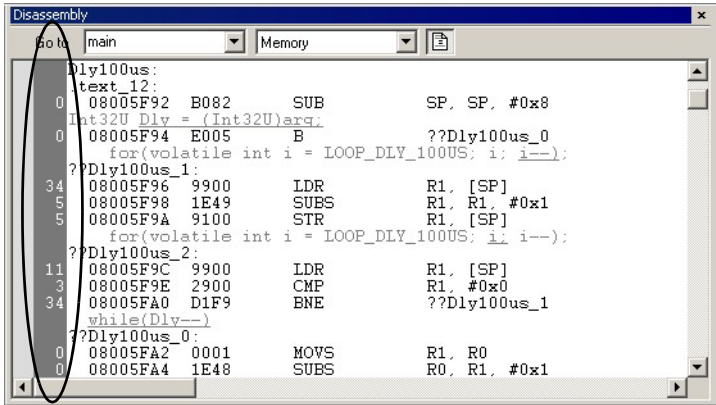


Figure 72: Instruction count in Disassembly window

For each instruction, the number of times it has been executed is displayed.

Instruction profiling attempts to use the same source as the function profiler. If the function profiler is not on, the instruction profiler will try to use first trace and then sampling as source. You can change the source to be used from the context menu that is available in the Function Profiler window.

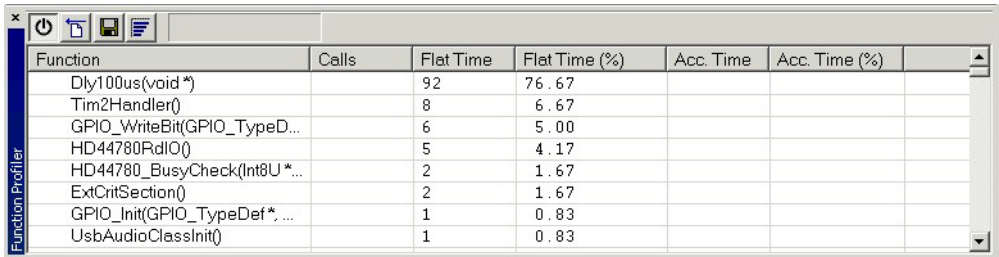
Reference information on the profiler

This section gives reference information about these windows and dialog boxes:

- *Function Profiler window*, page 157
- *Disassembly window*, page 69.

Function Profiler window

The Function Profiler window is available from the C-SPY driver menu.



Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
Dly100us(void *)	92	76 . 67			
Tim2Handler()	8	6 . 67			
GPIO_WriteBit(GPIO_TypeD...	6	5 . 00			
HD44780RdIO()	5	4 . 17			
HD44780_BusyCheck(Int8U *...	2	1 . 67			
ExtCritSection()	2	1 . 67			
GPIO_Init(GPIO_TypeDef *, ...	1	0 . 83			
UsbAudioClassInit()	1	0 . 83			





Figure 73: Function Profiler window

This window is available in the C-SPY simulator.

This window displays function profiling information.

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 154.

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 spent inside the function.
Flat time (%)	Trace (calls)	Flat time expressed as a percentage of the total time.
Acc. time	Trace (calls)	The time in cycles spent in this function and everything called by this function.
Acc. time (%)	Trace (calls)	Accumulated time in cycles 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:

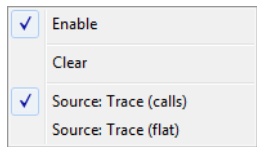


Figure 74: Function Profiler window context menu

These commands are available:

Enable	Enables the profiler. The system will collect information also when the window is closed.
Clear	Clears all profiling data.
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.

* The available sources depend on the C-SPY driver you are using.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 154.

Code coverage

This chapter describes the code coverage functionality in C-SPY®, which helps you verify whether all parts of your code have been executed. More specifically, this means:

- Introduction to code coverage
- Reference information on code coverage.

Introduction to code coverage

This section covers these topics:

- Reasons for using code coverage
- Briefly about code coverage
- Requirements 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 FOR USING CODE COVERAGE

Code coverage is not supported by all C-SPY drivers. For information about the driver you are using, see *Differences between the C-SPY drivers*, page 33. Code coverage is supported by the C-SPY Simulator.

Reference information on code coverage

This section gives reference information about these windows and dialog boxes:

- *Code Coverage window*, page 162.

See also *Single stepping*, page 64.

Code Coverage window

The Code Coverage window is available from the **View** menu.

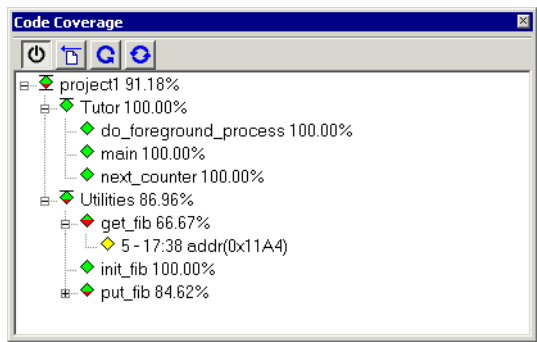


Figure 75: Code Coverage window

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:

- I 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

Table 12: Project options for enabling code coverage

Category	Setting
Debugger	Plugins>Code Coverage

Table 12: 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.

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 source 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:

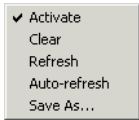








Figure 76: Code coverage window context menu

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.
	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.

Interrupts

This chapter describes how C-SPY® can help you test the logic of your interrupt service routines and debug the interrupt handling in the target system. More specifically, this chapter gives:

- Introduction to interrupts
- Procedures for interrupts
- Reference information on interrupts.

Introduction to interrupts

This section introduces you to interrupt logging and to interrupt simulation.

This section covers these topics:

- 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 187
- *Using breakpoints*, page 95
- *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 it has been serviced and is 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*.

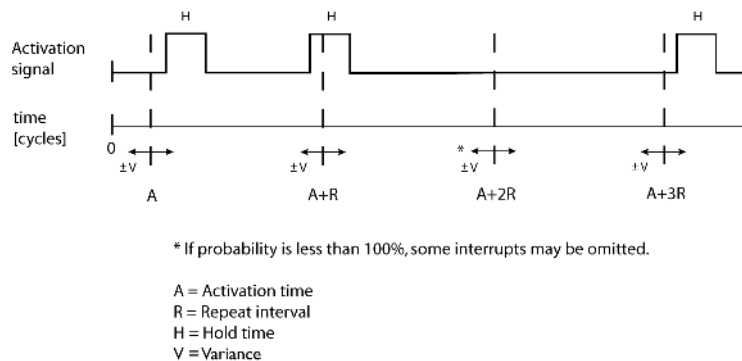


Figure 77: Simulated interrupt configuration

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 187.

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 C-SPY interrupt simulation has a simplified behavior compared to the hardware. This means that the execution of an interrupt is dependent on the status of the global interrupt enable bit.

To perform these actions for various devices, 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 49.

Procedures for interrupts

This section gives you step-by-step descriptions about interrupts.

More specifically, you will get information about:

- Simulating a simple interrupt.

See also:

- *Registering and executing using setup macros and setup files*, page 178 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 <stdio.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 = TIMER0_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>Interrupt Setup** to open the **Interrupts Setup** dialog box. Select the **Enable interrupt simulation** option to enable interrupt simulation. Click **New** to open the **Edit Interrupt** dialog box. For the Timer example, verify these settings:

Option	Settings
Interrupt	timer0 COMP
First activation	4000
Repeat interval	2000
Hold time	10
Probability (%)	100
Variance (%)	0

Table 13: Timer interrupt settings

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.
- 7 To watch the interrupt in action, choose **Simulator>Interrupt Log** to open the Interrupt Log window.
- 8 From the context menu, available in the Interrupt Log window, choose **Enable** to enable the logging. If you restart program execution, status information about entrances and exits to and from interrupts will now appear in the Interrupt Log window.

For information about how to get a graphical representation of the interrupts correlated with a time axis, see *Timeline window*, page 145.

Reference information on interrupts

This section gives reference information about these windows and dialog boxes:

- *Interrupts dialog box*, page 170.

Interrupts dialog box

The **Interrupts** dialog box is available by choosing **Simulator>Interrupts**.

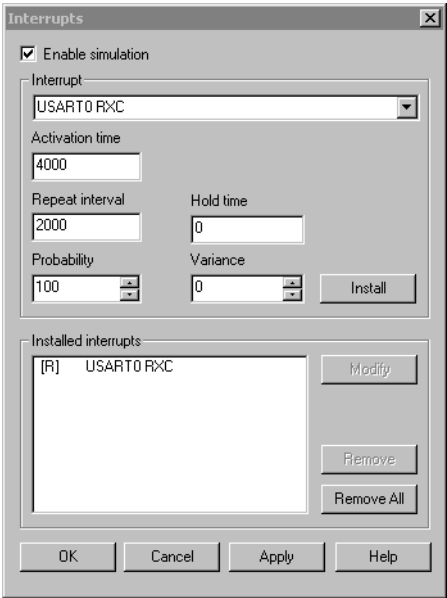


Figure 78: Interrupt Setup dialog box

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.

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:

Install	Installs the interrupt you specified.
Modify	Edits an existing interrupt.
Remove	Removes the selected interrupt.
Remove all	Removes all installed interrupts.

Using C-SPY macros

C-SPY® includes a comprehensive macro language which allows you to automate the debugging process and to simulate peripheral devices.

This chapter describes the C-SPY macro language, its features, for what purpose these features can be used, and how to use them. More specifically, this means:

- Introduction to C-SPY macros
- Procedures for 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.

Introduction to C-SPY macros

This section covers these topics:

- 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 `stack.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 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 186.

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 181.

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.

Procedures for using C-SPY macros

This section gives you step-by-step descriptions about how to register and execute C-SPY macros.

More specifically, you will get information about:

- Registering C-SPY macros—an overview
- Executing C-SPY macros—an overview
- Using the Macro Configuration dialog box
- Registering and executing using setup macros and setup files

- Executing macros using Quick Watch
- Executing a macro by connecting it to a breakpoint.

For more examples using C-SPY macros, see:

- The tutorial *Simulating an interrupt* in the Information Center
- *Initializing target hardware before C-SPY starts*, page 53.

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 macros interactively in the **Macro Configuration** dialog box, see *Using the Macro Configuration dialog box*, page 177.
- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files*, page 178.
- 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 201.

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 178.
- 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 179.
- 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 180.

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.

USING THE MACRO CONFIGURATION DIALOG BOX

The **Macro Configuration** dialog box is available by choosing **Debug>Macros**.

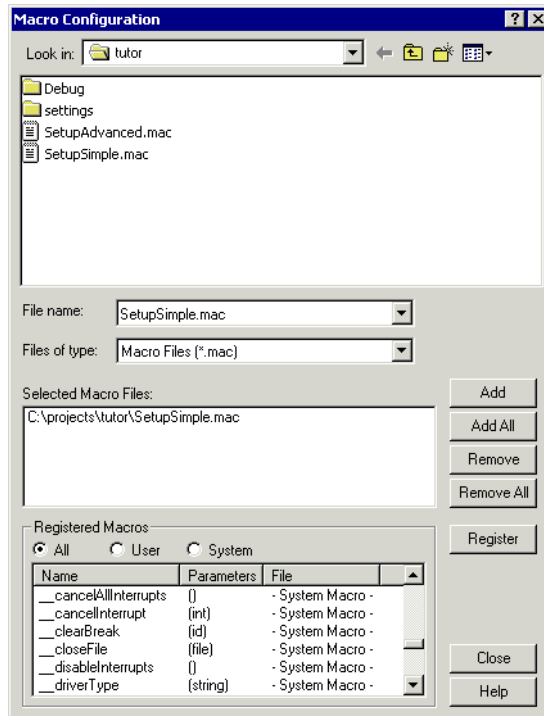


Figure 79: Macro Configuration dialog box

Use this dialog box to list, register, and edit your macro files and functions. The dialog box offers you an interactive interface for registering your macro functions which is convenient when you develop macro functions and continuously want to load and test them.

Macro functions that have been registered using the dialog box are deactivated when you exit the debug session, and will not automatically be registered at the next debug session.

To register a macro file:

- I Select the macro files you want to register in the file selection list, and click **Add** or **Add All** to add them to the **Selected Macro Files** list. Conversely, you can remove files from the **Selected Macro Files** list using **Remove** or **Remove All**.

- 2 Click **Register** to register the macro functions, replacing any previously defined macro functions or variables. Registered macro functions are displayed in the scroll list under **Registered Macros**.

Note: System macros cannot be removed from the list, they are always registered.

To list macro functions:

- 1 Select **All** to display all macro functions, select **User** to display all user-defined macros, or select **System** to display all system macros.
- 2 Click either **Name** or **File** under **Registered Macros** to display the column contents sorted by macro names or by file. Clicking a second time sorts the contents in the reverse order.

To modify a macro file:

Double-click a user-defined macro function in the **Name** column to open the file where the function is defined, allowing you to modify it.

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 debugger. 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 watchdog timer interrupt enable bit:

```
WDTstatus()
{
    if (#IE1 & 0x08 != 0) /* Checks the status of WDTIE */
        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 register the macro file, choose **Debug>Macros**. The **Macro Configuration** dialog box appears.
- 4 Locate the file, click **Add** and then **Register**. The macro function appears in the list of registered macros.
- 5 Choose **View>Quick Watch** to open the Quick Watch window, type the macro call `WDTstatus()` in the text field and press Return,

Alternatively, in the macro file editor window, select the macro function name `WDTstatus()`. Right-click, and choose **Quick Watch** from the context menu that appears.

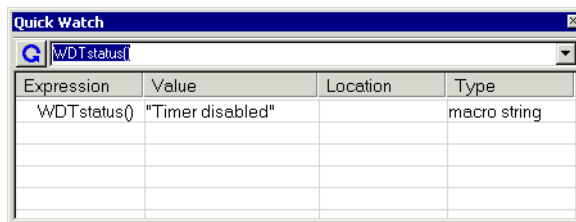


Figure 80: Quick Watch window

The macro will automatically be displayed in the Quick Watch window.

For more information, see *Quick Watch window*, page 92.

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 **Debug>Macros** to open the **Macro Configuration** dialog box and add your macro file to the list **Selected Macro Files**. Click **Register** and your macro function will appear in the list **Registered Macros**. Close the dialog box.
- 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 111
 - Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 105.
- 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 184.

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.

Reference information on the macro language

This section gives reference information on the macro language:

- *Macro functions*, page 181
- *Macro variables*, page 182
- *Macro strings*, page 182
- *Macro statements*, page 183
- *Formatted output*, page 184.

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 82.

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
myvar = 3.5;	myvar is now type float, value 3.5.
myvar = (int*)i;	myvar is now type pointer to int, and the value is the same as i.

Table 14: 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 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 184.

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 82.

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 Debug Log 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 196.

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 174.

This table summarizes the reserved setup macro function names:

Macro	Description
<code>execUserPreload</code>	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.
<code>execUserSetup</code>	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.
<code>execUserPreReset</code>	Called each time just before the reset command is issued. Implement this macro to set up any required device state.
<code>execUserReset</code>	Called each time just after the reset command is issued. Implement this macro to set up and restore data.
<code>execUserExit</code>	Called once when the debug session ends. Implement this macro to save status data etc.

Table 15: C-SPY setup macros



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 *Simulating an interrupt* 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.

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>
<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>__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>__memoryRestoreFromFile</code>	Reads from a file and restores to memory
<code>__memorySave</code>	Saves the contents of a specified memory area to a file
<code>__memorySaveToFile</code>	Saves a range of a memory zone to a file
<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

Table 16: Summary of system macros

Macro	Description
__resetFile	Rewinds a file opened by __openFile
__setCodeBreak	Sets a code breakpoint
__setComplexBreak	Sets a complex breakpoint
__setDataBreak	Sets a data breakpoint
__setLogBreak	Sets a log breakpoint
__setSimBreak	Sets a simulation breakpoint
__setTraceStartBreak	Sets a trace start breakpoint
__setTraceStopBreak	Sets a trace stop breakpoint
__sourcePosition	Returns the file name and source location if the current execution location corresponds to a source location
__strFind	Searches a given string for the occurrence of another string
__subString	Extracts a substring from another string
__targetDebuggerVersion	Returns the version of the target debugger
__toLower	Returns a copy of the parameter string where all the characters have been converted to lower case
__toString	Prints strings
__toUpper	Returns a copy of the parameter string where all the characters have been converted to upper case
__transparent	Sends a command to the ROM-monitor transparently from C-SPY
__unloadImage	Unloads a debug image.
__writeFile	Writes to the specified file
__writeFileByte	Writes one byte to the specified file
__writeMemory8, __writeMemoryByte	Writes one byte to the specified memory location
__writeMemory16	Writes a two-byte word to the specified memory location
__writeMemory32	Writes a four-byte word to the specified memory location

Table 16: Summary of system macros (Continued)

__cancelAllInterrupts

Syntax	<code>__cancelAllInterrupts()</code>
Return value	<code>int 0</code>
Description	Cancels all ordered interrupts.
Applicability	This system macro is only available in the C-SPY Simulator.

__cancelInterrupt

Syntax	<code>__cancelInterrupt(<i>interrupt_id</i>)</code>
Parameter	<div><div><code><i>interrupt_id</i></code></div><div>The value returned by the corresponding <code>__orderInterrupt</code> macro call (unsigned long)</div></div>

Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td><code>int 0</code></td></tr><tr><td>Unsuccessful</td><td>Non-zero error number</td></tr></table>	Result	Value	Successful	<code>int 0</code>	Unsuccessful	Non-zero error number
Result	Value						
Successful	<code>int 0</code>						
Unsuccessful	Non-zero error number						

Table 17: `__cancelInterrupt` return values

Description	Cancels the specified interrupt.
Applicability	This system macro is only available in the C-SPY Simulator.

__clearBreak

Syntax	<code>__clearBreak(<i>break_id</i>)</code>
Parameter	<div><div><code><i>break_id</i></code></div><div>The value returned by any of the set breakpoint macros</div></div>
Return value	<code>int 0</code>
Description	Clears a user-defined breakpoint.
See also	<i>Using breakpoints</i> , page 95.

__closeFile

Syntax	<code>__closeFile(<i>fileHandle</i>)</code>	
Parameter	<i>fileHandle</i>	The macro variable used as filehandle by the <code>__openFile</code> macro
Return value	<code>int 0</code>	
Description	Closes a file previously opened by <code>__openFile</code> .	

__delay

Syntax	<code>__delay(<i>value</i>)</code>	
Parameter	<i>value</i>	The number of milliseconds to delay execution
Return value	<code>int 0</code>	
Description	Delays execution the specified number of milliseconds.	

__disableInterrupts

Syntax	<code>__disableInterrupts()</code>							
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td>int 0</td></tr><tr><td>Unsuccessful</td><td>Non-zero error number</td></tr></table>		Result	Value	Successful	int 0	Unsuccessful	Non-zero error number
Result	Value							
Successful	int 0							
Unsuccessful	Non-zero error number							

Table 18: `__disableInterrupts` return values

Description	Disables the generation of interrupts.
Applicability	This system macro is only available in the C-SPY Simulator.

__driverType

Syntax `__driverType(driver_id)`

Parameter *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 JTAGICE driver.
- "jtagicemkII" corresponds to the JTAGICE mkII driver.
- "jtagice3" corresponds to the JTAGICE3 driver.
- "avrone" corresponds to the AVR ONE! driver.
- "ice200" corresponds to the ICE200 driver.
- "ccr" corresponds to the CCR driver.

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 19: __driverType return values

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
Unsuccessful	Non-zero error number

Table 20: __enableInterrupts return values

Description Enables the generation of interrupts.

Applicability This system macro is only available in the C-SPY Simulator.

__evaluate

Syntax

`__evaluate(string, valuePtr)`

Parameter

`string`

Expression string

`valuePtr`

Pointer to a macro variable storing the result

Return value

Result	Value
Successful	int 0
Unsuccessful	int 1

Table 21: __evaluate return values

Description

This macro interprets the input string as an expression and evaluates it. The result is stored in a variable pointed to by `valuePtr`.

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.

__getCycleCounter

Syntax

`__getCycleCounter()`

Return value

Returns the current value of the cycle counter as a long long int.

Description

Reads the current value of the cycle counter.

__isBatchMode

Syntax

`__isBatchMode()`

Return value

Result	Value
True	int 1
False	int 0

Table 22: __isBatchMode return values

Description This macro returns True if the debugger is running in batch mode, otherwise it returns False.

__loadImage

Syntax `__loadImage(path, offset, debugInfoOnly)`

Parameter	<i>path</i>	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 <i>IDE Project Management and Building Guide</i> .
	<i>offset</i>	An integer that identifies the offset to the destination address for the downloaded image.
	<i>debugInfoOnly</i>	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 23: __loadImage return values

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 241 and *Loading multiple images*, page 51.

__memoryRestore

Syntax	<code>__memoryRestore(zone, filename)</code>	
Parameters	<i>zone</i>	The memory zone name (string); for a list of available zones, see <i>C-SPY memory zones</i> , page 122.
	<i>filename</i>	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 <i>IDE Project Management and Building Guide</i> .
Return value	0 if successful, otherwise 1	
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 <code>__memoryRestoreFromFile</code> .	
Example	<code>__memoryRestore("DATA", "c:\\temp\\saved_memory.hex");</code>	
See also	<i>Memory Restore dialog box</i> , page 129.	

__memoryRestoreFromFile

Syntax	<code>__memoryRestoreFromFile(filename, zone)</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 122.

Return value	0 if successful, otherwise 1
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	<pre>__memoryRestoreFromFile("C:\\temp\\tmp.hex", "DATA");</pre>

__memorySave

Syntax	<code>__memorySave(start, stop, format, filename)</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: intel-extended motorola motorola-s19 motorola-s28 motorola-s37.	
	<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.		
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	<code>__memorySave("DATA:0x1000", "DATA:0x1100", "intel-extended", "c:\\temp\\saved_memory.hex");</code>		
See also	<i>Memory Save dialog box</i> , page 128.		

__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 122.
	<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	
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>	

__openFile

Syntax	<code>__openFile(filename, access)</code>	
Parameters	<i>filename</i>	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 <i>IDE Project Management and Building Guide</i> .

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
"w" write

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 24: __openFile return values

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	<code>__orderInterrupt(<i>specification</i>, <i>first_activation</i>, <i>repeat_interval</i>, <i>variance</i>, <i>hold_time</i>, <i>probability</i>)</code>	
Parameters	<i>specification</i>	The interrupt name (string). The interrupt system will automatically get the description from the device description file.
	<i>first_activation</i>	The first activation time in cycles (integer)
	<i>repeat_interval</i>	The periodicity in cycles (integer)
	<i>variance</i>	The timing variation range in percent (integer between 0 and 100)
	<i>hold_time</i>	The hold time (integer)
	<i>probability</i>	The probability in percent (integer between 0 and 100)
Return value	The macro returns an interrupt identifier (unsigned long). If the syntax of <i>specification</i> is incorrect, it returns -1.	
Description	Generates an interrupt.	
Applicability	This system macro is only available in the C-SPY Simulator.	
Example	This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles: <code>__orderInterrupt("USART0_TXC", 4000, 2000, 0, 0, 100);</code>	

__readFile

Syntax	<code>__readFile(<i>fileHandle</i>, <i>valuePtr</i>)</code>	
Parameters	<i>fileHandle</i>	A macro variable used as filehandle by the <code>__openFile</code> macro
	<i>valuePtr</i>	A pointer to a variable

Return value

Result	Value
Successful	0
Unsuccessful	Non-zero error number

Table 25: `__readFile` return values

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.

Example

```
__var number;
if (__readFile(myFileHandle, &number) == 0)
{
    // Do something with number
}
```

`__readFileByte`

Syntax

```
__readFileByte(fileHandle)
```

Parameter

<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.

Description

Reads one byte from a file.

Example

```
__var byte;
while ( (byte = __readFileByte(myFileHandle)) != -1 )
{
    /* Do something with byte */
}
```

`__readMemory8`, `__readMemoryByte`

Syntax

```
__readMemory8(address, zone)
__readMemoryByte(address, zone)
```

Parameters

<i>address</i>	The memory address (integer)
----------------	------------------------------

	<i>zone</i>	The memory zone name (string); for more information about available zones, see <i>C-SPY memory zones</i> , page 122.
Return value	The macro returns the value from memory.	
Description	Reads one byte from a given memory location.	
Example	<code>__readMemory8(0x0108, "DATA");</code>	

__readMemory16

Syntax	<code>__readMemory16(address, zone)</code>	
Parameters	<i>address</i>	The memory address (integer)
	<i>zone</i>	The memory zone name (string); for more information about available zones, see <i>C-SPY memory zones</i> , page 122.
Return value	The macro returns the value from memory.	
Description	Reads a two-byte word from a given memory location.	
Example	<code>__readMemory16(0x0108, "DATA");</code>	

__readMemory32

Syntax	<code>__readMemory32(address, zone)</code>	
Parameters	<i>address</i>	The memory address (integer)
	<i>zone</i>	The memory zone name (string); for more information about available zones, see <i>C-SPY memory zones</i> , page 122.
Return value	The macro returns the value from memory.	
Description	Reads a four-byte word from a given memory location.	
Example	<code>__readMemory32(0x0108, "DATA");</code>	

__registerMacroFile

Syntax	<code>__registerMacroFile(filename)</code>	
Parameter	<i>filename</i>	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> .
Return value	<code>int 0</code>	
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>Registering and executing using setup macros and setup files</i> , page 178.	

__resetFile

Syntax	<code>__resetFile(fileHandle)</code>	
Parameter	<i>fileHandle</i>	A macro variable used as filehandle by the <code>__openFile</code> macro
Return value	<code>int 0</code>	
Description	Rewinds a file previously opened by <code>__openFile</code> .	

__setCodeBreak

Syntax	__setCodeBreak(<i>location</i> , <i>count</i> , <i>condition</i> , <i>cond_type</i> , <i>action</i>)	
Parameters	<i>location</i>	<p>A string with a location description. Choose between:</p> <p>A C-SPY expression, whose value evaluates to a valid address, such as a function, for example <code>main</code>. For more information about C-SPY expressions, see <i>C-SPY expressions</i>, page 76.</p> <p>An absolute address, on the form <i>zone:hexaddress</i> or simply <i>hexaddress</i> (for example <code>Memory:42</code>). <i>zone</i> refers to C-SPY memory zones and specifies in which memory the address belongs.</p> <p>A source location in your C source code, using the syntax <code>{filename}.row.col</code>. For example <code>{D:\src\prog.c}.22.3</code> sets a breakpoint on the third character position on row 22 in the source file <code>prog.c</code>. Note that the Source location type is usually meaningful only for code breakpoints.</p>
	<i>count</i>	The number of times that a breakpoint condition must be fulfilled before a break occurs (integer)
	<i>condition</i>	The breakpoint condition (string)
	<i>cond_type</i>	The condition type; either “CHANGED” or “TRUE” (string)
	<i>action</i>	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 26: __setCodeBreak return values		
Description	Sets a code breakpoint, that is, a breakpoint which is triggered just before the processor fetches an instruction at the specified location.	
Examples	__setCodeBreak("D:\src\prog.c".12.9", 3, "d>16", "TRUE", "ActionCode()");	

This example sets a code breakpoint on the label `main` in your source:

```
__setCodeBreak("main", 0, "1", "TRUE", "");
```

See also

Using breakpoints, page 95.

__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 <code>{filename}.line.col</code>, for example <code>{D:\\src\\prog.c}.12.9</code>, although this is not very useful for data breakpoints.</p> <p>An absolute location on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code>, for example <code>Memory:0x42</code></p> <p>An expression whose value designates a location, for example <code>myGlobalVariable</code>.</p>
<i>b_addr</i>	<p>A string with a location description. This can be:</p> <p>A source location on the form <code>{filename}.line.col</code>, for example <code>{D:\\src\\prog.c}.12.9</code>, although this is not very useful for data breakpoints.</p> <p>An absolute location on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code>, for example <code>Memory:0x42</code>.</p> <p>An expression whose value designates a location, for example <code>myGlobalVariable</code>.</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>	Single-byte value for comparison with the memory contents of <i>a_addr</i> and <i>b_addr</i> .
<i>d_value</i>	Single-byte value for comparison with the memory contents of <i>a_addr</i> and <i>b_addr</i> .
<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>	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: __set Complex Break return values

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.

Applicability

This macro is only available for the C-SPY driver for AVR ONE! and the C-SPY driver for JTAGICE3.

Examples

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

Using breakpoints, page 95.

__setDataBreak

Syntax

```
__setDataBreak(location, count, condition, cond_type, access,  
              action)
```

Parameters

location

A string with a location description. Choose between:

A C-SPY expression, whose value evaluates to a valid address, such as a variable name. For example, `my_var` refers to the location of the variable `my_var`, and `arr[3]` refers to the location of the third 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 76.

An absolute address, on the form `zone:hexaddress` or simply `hexaddress` (for example `Memory:42`). *zone* refers to C-SPY memory zones and specifies in which memory the address belongs.

A source location in your C source code, using the syntax `{filename}.row.col`. For example `{D:\src\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.

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 28: `__setDataBreak` return values

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.

Applicability

This system macro is only available in the C-SPY Simulator, for the C-SPY driver for AVR ONE!, and for the C-SPY driver for JTAGICE3.

Example

```
__var brk;
brk = __setDataBreak("DATA:0x110", 3, "d>6", "TRUE",
    "W", "ActionData()");
...
__clearBreak(brk);
```

See also

Using breakpoints, page 95.

`__setLogBreak`

Syntax

```
__setLogBreak(location, message, mesg_type, condition,
    cond_type)
```

Parameters

- location*
- A string with a location description. Choose between:
 - A C-SPY expression, whose value evaluates to a valid address, such as a function, for example `main`. For more information about C-SPY expressions, see *C-SPY expressions*, page 76.
 - An absolute address, on the form *zone:hexaddress* or simply *hexaddress* (for example `Memory:42`). *zone* refers to C-SPY memory zones and specifies in which memory the address belongs.
 - A source location in your C source code, using the syntax `{filename}.row.col`. For example `{D:\src\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.

<i>message</i>	The message text
<i>msg_type</i>	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.
<i>condition</i>	The breakpoint condition (string)
<i>cond_type</i>	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 29: `__setLogBreak` return values

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 184 and *Using breakpoints*, page 95.

__setSimBreak

Syntax

__setSimBreak(*location*, *access*, *action*)

Parameters

<i>location</i>	<p>A string with a location description. Choose between:</p> <p>A C-SPY expression, whose value evaluates to a valid address, such as a variable name. For example, <code>my_var</code> refers to the location of the variable <code>my_var</code>, and <code>arr[3]</code> refers to the location of the third element of the array <code>arr</code>. For static variables declared with the same name in several functions, use the syntax <code>my_func::my_static_variable</code> to refer to a specific variable. For more information about C-SPY expressions, see <i>C-SPY expressions</i>, page 76.</p> <p>An absolute address, on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code> (for example <code>Memory:42</code>). <i>zone</i> refers to C-SPY memory zones and specifies in which memory the address belongs.</p> <p>A source location in your C source code, using the syntax <code>{filename}.row.col</code>. For example <code>{D:\src\prog.c}.22.3</code> sets a breakpoint on the third character position on row 22 in the source file <code>prog.c</code>. Note that the Source location type is usually meaningful only for code breakpoints.</p>
<i>access</i>	<p>The memory access type: "R" for read or "W" for write</p>
<i>action</i>	<p>An expression, typically a call to a macro function, 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 30: __setSimBreak return values

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.

Applicability This system macro is only available in the C-SPY Simulator.

__setTraceStartBreak

Syntax __setTraceStartBreak(*location*)

Parameters

location A string with a location description. Choose between:

 A C-SPY expression, whose value evaluates to a valid address, such as a function, for example `main`. For more information about C-SPY expressions, see *C-SPY expressions*, page 76.

 An absolute address, on the form *zone:hexaddress* or simply *hexaddress* (for example `Memory:42`). *zone* refers to C-SPY memory zones and specifies in which memory the address belongs.

 A source location in your C source code, using the syntax `{filename}.row.col`. For example `{D:\\src\\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.

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 31: __setTraceStartBreak return values

Description Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is started.

Applicability This system macro is only available in the C-SPY Simulator.

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

Using breakpoints, page 95.

__setTraceStopBreak**Syntax**

```
__setTraceStopBreak(location)
```

Parameters

location

A string with a location description. Choose between:

A C-SPY expression, whose value evaluates to a valid address, such as a function, for example `main`. For more information about C-SPY expressions, see *C-SPY expressions*, page 76.

An absolute address, on the form *zone:hexaddress* or simply *hexaddress* (for example `Memory:42`). *zone* refers to C-SPY memory zones and specifies in which memory the address belongs.

- A source location in your C source code, using the syntax `{filename}.row.col`. For example `{D:\\src\\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that the Source location type is usually meaningful only for code breakpoints.

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 32: `__setTraceStopBreak` return values

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is stopped.

Applicability

This system macro is only available in the C-SPY Simulator.

Example

See `__setTraceStartBreak`, page 210.

See also

Using breakpoints, page 95.

`__sourcePosition`

Syntax

`__sourcePosition(linePtr, colPtr)`

Parameters

<i>linePtr</i>	Pointer to the variable storing the line number
<i>colPtr</i>	Pointer to the variable storing the column number

Return value

Result	Value
Successful	Filename string
Unsuccessful	Empty (" ") string

Table 33: `__sourcePosition` return values

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

<i>macroString</i>	The macro string to search in
--------------------	-------------------------------

<i>pattern</i>	The string pattern to search for
<i>position</i>	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.
Description	This macro searches a given string for the occurrence of another string.
Example	<pre>__strFind("Compiler", "pile", 0) = 3 __strFind("Compiler", "foo", 0) = -1</pre>
See also	<i>Macro strings</i> , page 182.

__subString

Syntax	<code>__subString(<i>macroString</i>, <i>position</i>, <i>length</i>)</code>
Parameters	<p><i>macroString</i> The macro string from which to extract a substring</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.
Description	This macro extracts a substring from another string.
Example	<pre>__subString("Compiler", 0, 2)</pre> <p>The resulting macro string contains Co.</p> <pre>__subString("Compiler", 3, 4)</pre> <p>The resulting macro string contains pile.</p>
See also	<i>Macro strings</i> , page 182.

__targetDebuggerVersion

Syntax	<code>__targetDebuggerVersion</code>
Return value	A string that represents the version number of the C-SPY debugger processor module.

Description	This macro returns the version number of the C-SPY debugger processor module.	
Example	<pre>__var toolVer; toolVer = __targetDebuggerVersion(); __message "The target debugger version is, ", toolVer;</pre>	
__toLower		
Syntax	<pre>__toLower(<i>macroString</i>)</pre>	
Parameter	<i>macroString</i>	Any macro string
Return value	The converted macro string.	
Description	This macro returns a copy of the parameter string where all the characters have been converted to lower case.	
Example	<pre>__toLower("IAR") The resulting macro string contains iar. __toLower("Mix42") The resulting macro string contains mix42.</pre>	
See also	<i>Macro strings</i> , page 182.	
__toString		
Syntax	<pre>__toString(<i>C_string</i>, <i>maxlength</i>)</pre>	
Parameter	<i>C_string</i>	Any null-terminated C string
	<i>maxlength</i>	The maximum length of the returned macro string
Return value	Macro string.	
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:

```
__toString(hptr, 5)
```

would return the macro string containing Hello.

See also

Macro strings, page 182.

__toUpper

Syntax

```
__toUpper(macroString)
```

Parameter

macroString is any macro string.

Return value

The converted string.

Description

This macro returns a copy of the parameter *macroString* where all the characters have been converted to upper case.

Example

```
__toUpper("string")
```

The resulting macro string contains STRING.

See also

Macro strings, page 182.

__transparent

Syntax

```
__transparent(commandstring)
```

Parameter

commandstring The command to send to the ROM-monitor.

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.

Applicability

This system macro is only available in the CCR driver.

See also

Using C-SPY macros for transparent commands, page 285.

__unloadImage

Syntax

__unloadImage(module_id)

Parameter

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
module_id	A unique module identification (the same as the input parameter).
int 0	The unloading failed.

Table 34: __unloadImage return values

Description

Unloads debug information from an already downloaded image.

See also

Loading multiple images, page 51 and Images, page 241.

__writeFile

Syntax	<code>__writeFile(fileHandle, value)</code>	
Parameters	<i>fileHandle</i>	A macro variable used as filehandle by the <code>__openFile</code> macro
	<i>value</i>	An integer
Return value	<code>int 0</code>	
Description	Prints the integer value in hexadecimal format (with a trailing space) to the file <i>file</i> . Note: The <code>__fmessage</code> statement can do the same thing. The <code>__writeFile</code> macro is provided for symmetry with <code>__readFile</code> .	

__writeFileByte

Syntax	<code>__writeFileByte(<i>fileHandle</i>, <i>value</i>)</code>	
Parameters	<i>fileHandle</i>	A macro variable used as filehandle by the <code>__openFile</code> macro
	<i>value</i>	An integer in the range 0-255
Return value	<code>int 0</code>	
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	<i>value</i>	The value to be written (integer)
	<i>address</i>	The memory address (integer)
	<i>zone</i>	The memory zone name (string); for more information about available zones, see <i>C-SPY memory zones</i> , page 122.
Return value	<code>int 0</code>	
Description	Writes one byte to a given memory location.	
Example	<code>__writeMemory8(0x2F, 0x8020, "DATA");</code>	

__writeMemory16

Syntax	<code>__writeMemory16(<i>value</i>, <i>address</i>, <i>zone</i>)</code>	
Parameters	<i>value</i>	The value to be written (integer)
	<i>address</i>	The memory address (integer)

	<i>zone</i>	The memory zone name (string); for more information about available zones, see <i>C-SPY memory zones</i> , page 122.
Return value	int 0	
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	<i>value</i>	The value to be written (integer)
	<i>address</i>	The memory address (integer)
	<i>zone</i>	The memory zone name (string); for more information about available zones, see <i>C-SPY memory zones</i> , page 122.
Return value	int 0	
Description	Writes four bytes to a given memory location.	
Example	<code>__writeMemory32(0x5555FFFF, 0x8020, "DATA");</code>	

The C-SPY Command Line Utility—cspybat

This chapter describes how you can execute C-SPY® in batch mode, using the C-SPY Command Line Utility—cspybat.exe. More specifically, this means:

- 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`.

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>).
<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 223.

Table 35: *cspybat* parameters

Parameter	Description
--backend	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.
driver_options	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 223.

Table 35: cspybat parameters (Continued)

Example

This example starts cspybat using the simulator driver:

```
EW_DIR\common\bin\cspybat EW_DIR\avr\bin\avrproc.dll
EW_DIR\avr\bin\avr\sim.dll PROJ_DIR\myproject.d90 --plugin
EW_DIR\avr\bin\avrLibSupportbat.dll --backend -d sim --cpu=m128
-p EW_DIR\avr\bin\config\iom128.ddf
```

where *EW_DIR* is the full path of the directory where you have installed IAR Embedded Workbench

and where *PROJ_DIR* is the path of your project directory.

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 235.
- Error return codes
`cspybat` return 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.

USING AN AUTOMATICALLY GENERATED BATCH FILE

When you use C-SPY in the IDE, C-SPY generates a batch file *projectname.cspy.bat* every time C-SPY is initialized. You can find the file in the

directory \$PROJ_DIR\$\settings. This batch file contains the same settings as in the IDE, and with minimal modifications, you can use it from the command line to start cspybat. The file also contains information about required modifications.

Summary of C-SPY command line options

These options are available:

C-SPY command line option	Simulator	JTAGICE	JTAGICE mkII	Dragon	JTAGICE3	AVR ONE!	ICE200	CCR
--64bit_doubles	x	x	x	x	x	x	x	x
--64k_flash	x	x	x	x	x	x	x	x
--avrone_jtag_clock	--	--	--	--	x	x	--	--
--backend	x	x	x	x	x	x	x	x
--code_coverage_file	x	x	x	x	x	x	x	x
--cpu	x	x	x	x	x	x	x	x
--cycles	x	x	x	x	x	x	x	x
-d	x	x	x	x	x	x	x	x
--disable_internal_e eprom	x	x	x	x	x	x	x	x
--disable_interrupts	x	--	--	--	--	--	--	--
--download_only	x	x	x	x	x	x	x	x
--drv_communication	--	x	x	x	x	x	x	x
--drv_communication_ log	--	x	x	x	x	x	x	x
--drv_debug_port	--	--	--	--	x	x	--	--
--drv_download_data	--	x	x	x	x	x	x	x
--drv_dragon	--	--	--	x	--	--	--	--
--drv_set_exit_break point	--	x	x	x	x	x	--	--
--drv_set_getchar_br eakpoint	--	x	x	x	x	x	--	--
--drv_set_putchar_br eakpoint	--	x	x	x	x	x	--	--
--drv_suppress_downl oad	--	x	x	x	x	x	x	x

Table 36: C-SPY command line options

C-SPY command line option	Simulator	JTAGICE	JTAGICE mkII	Dragon	JTAGICE3	AVR ONE!	ICE200	CCR
--drv_use_PDI	--	--	x	x	--	--	--	--
--drv_verify_downloa d	--	x	x	x	x	x	x	x
--eeprom_size	x	x	x	x	x	x	x	x
--enhanced_core	x	x	x	x	x	x	x	x
--ice200_restore_EEP ROM	--	--	--	--	--	--	x	--
--ice200_single_step _timers	--	--	--	--	--	--	x	--
--jtagice_clock	--	x	x	x	--	--	--	--
--jtagice_do_hardwar e_reset	--	x	x	x	x	x	--	--
--jtagice_leave_time rs_running	--	x	x	x	x	x	--	--
--jtagice_preserve_e eprom	--	x	x	x	x	x	--	--
--jtagice_restore_fu se	--	x	x	x	x	x	--	--
--jtagicemkII_use_so ftware_breakpoints	--	--	x	x	--	--	--	--
--macro	x	x	x	x	x	x	x	x
-p	x	x	x	x	x	x	x	x
--plugin	x	x	x	x	x	x	x	x
--silent	x	x	x	x	x	x	x	x
--timeout	x	x	x	x	x	x	x	x
-v	x	x	x	x	x	x	x	x

Table 36: C-SPY command line options

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	<code>--64bit_doubles</code>
Applicability	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	<code>--64k_flash</code>
Applicability	All C-SPY drivers.
Description	Use this option to enable 64-Kbytes flash mode for the processor configurations <code>-v2</code> and <code>-v3</code> .



Project>Options>General Options>Target>No RAMPZ register

--avrone_jtag_clock

Syntax	<code>--avrone_jtag_clock=speed</code>
Parameters	<p><i>speed</i> The JTAG or PDI clock frequency in Hz. Possible values are 0-65535000 Hz in steps of 1000 Hz.</p>
Applicability	The C-SPY AVR ONE! and JTAGICE3 drivers.
Description	Use this option to specify the speed of the debugging interface.



Project>Options>Debugger>Driver>Driver 1>Debug Port>Frequency in kHz

--backend

Syntax	<code>--backend {driver options}</code>	
Parameters	<code>driver options</code>	Any option available to the C-SPY driver you are using.
Applicability	Sent to <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.	

--cpu

Syntax	<code>--cpu=cpu_name</code>	
Parameters	<code>cpu_name</code>	The CPU model, 2323, 8515, m103, etc.
Applicability	All C-SPY drivers.	
Description	Use this option to specify the CPU model your application was compiled for. This option cannot be used together with <code>-v</code> .	



Project>Options>General Options>Target>Processor configuration

--code_coverage_file

Syntax	<code>--code_coverage_file file</code>	
Parameters	<code>file</code>	The name of the destination file for the code coverage information.
Applicability	Sent to <code>cspybat</code> .	
Description	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.	

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 `stderr`.

See also *Code coverage*, page 161.


--cycles

Syntax	<code>--cycles <i>cycles</i></code>
Parameters	<i>cycles</i> The number of cycles to run.
Applicability	Sent to <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.


-d

Syntax	<code>-d {<i>driver</i> <i>thirdpartydriver</i>}</code>
Parameters	<i>driver</i> Specifies the C-SPY driver. you are using. Choose between: <ul style="list-style-type: none"> <code>avrone</code> <code>ccr</code> <code>ice200</code> <code>jtagice</code> <code>jtagicemkII</code> <code>jtagice3</code> <code>sim</code>
Applicability	All C-SPY drivers.
Description	Use this option to specify the C-SPY driver to be used.


--disable_internal_eeprom

Syntax	<code>--disable_internal_eeprom</code>
Applicability	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	<code>--disable_interrupts</code>
Applicability	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	<code>--download_only</code>
Applicability	Sent to <code>cspybat</code> .
Description	Use this option to download the code image without starting a debug session afterwards.
	 To set related options, choose: Project>Download

--drv_communication

Syntax	<code>--drv_communication=[COMn USB]</code>		
Parameters	<table><tr><td>COMn</td><td>A serial communication port. <i>n</i> can be between 1 and 32. Note that COMn is not used in AVR ONE! and JTAGICE3.</td></tr></table>	COMn	A serial communication port. <i>n</i> can be between 1 and 32. Note that COMn is not used in AVR ONE! and JTAGICE3.
COMn	A serial communication port. <i>n</i> can be between 1 and 32. Note that COMn is not used in AVR ONE! and JTAGICE3.		

	USB	The USB port. Can only be used by the AVR ONE!, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.
Applicability	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	<code>--drv_communication_log=<i>filename</i></code>	
Parameters	<i>filename</i>	The name of the log file.
Applicability	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>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	<i>autodetect</i>	Specifies auto-detection of the debugging interface.
	<i>debugwire</i>	Specifies the debugWIRE debugging interface.

	pdi	Specifies the PDI debugging interface.
	jtag	Specifies the JTAG debugging interface.

Applicability The C-SPY AVR ONE! and JTAGICE3 drivers.

Description Use this option to specify the debug interface.



Project>Options>Debugger>AVR ONE!>AVR ONE! 1>Debug Port

--drv_download_data

Syntax --drv_download_data

Applicability All C-SPY hardware drivers.

Description Use this option to enable downloading of constant data into RAM.



Project>Options>Debugger>AVR ONE!>AVR ONE! 1>Allow download to RAM

Project>Options>Debugger>CCR>CCR>Allow download to RAM

Project>Options>Debugger>ICE200>ICE200>Allow download to RAM

Project>Options>Debugger>JTAGICE>JTAGICE 1>Allow download to RAM

Project>Options>Debugger>JTAGICE3>JTAGICE3 1>Allow download to RAM

Project>Options>Debugger>JTAGICE mkII>JTAGICE mkII 1>Allow download to RAM

Project>Options>Debugger>Dragon>Dragon 1>Allow download to RAM

--drv_dragon

Syntax --drv_dragon

Applicability The C-SPY AVR Dragon driver.

Description Use this option to specify the AVR Dragon driver to be used.



Project>Options>Debugger>Driver

--drv_set_exit_breakpoint

Syntax `--drv_set_exit_breakpoint`

Applicability The C-SPY AVR ONE!, JTAGICE, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.

Description Use this option in the CLIB runtime environment to set a system breakpoint on the `exit` label. This consumes a hardware breakpoint.

See also *Breakpoint consumers*, page 99



Project>Options>Debugger>Driver>Driver 2>System breakpoints on>exit

--drv_set_getchar_breakpoint

Syntax `--drv_set_getchar_breakpoint`

Applicability The C-SPY AVR ONE!, JTAGICE, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.

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 99



Project>Options>Debugger>Driver>Driver 2>System breakpoints on>getchar

--drv_set_putchar_breakpoint

Syntax `--drv_set_putchar_breakpoint`

Applicability The C-SPY AVR ONE!, JTAGICE, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.

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 99



Project>Options>Debugger>Driver>Driver 2>System breakpoints on>putchar

--drv_suppress_download

Syntax	<code>--drv_suppress_download</code>
Applicability	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>AVR ONE!>AVR ONE! 1>Suppress download
Project>Options>Debugger>CCR>CCR>Suppress download
Project>Options>Debugger>ICE200>ICE200>Suppress download
Project>Options>Debugger>JTAGICE>JTAGICE 1>Suppress download
Project>Options>Debugger>JTAGICE3>JTAGICE3 1>Suppress download
Project>Options>Debugger>JTAGICE mkII>JTAGICE mkII 1>Suppress download
Project>Options>Debugger>Dragon>Dragon 1>Suppress download

--drv_use_PDI

Syntax	<code>--drv_use_PDI</code>
Applicability	The C-SPY JTAGICE mkII and AVR Dragon drivers.
Description	Use this option if you want the C-SPY driver to communicate with the device using the PDI interface.



Project>Options>Debugger>JTAGICE mkII>JTAGICE mkII 1>Use PDI

--drv_verify_download

Syntax	<code>--drv_verify_download</code>
Applicability	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>AVR ONE!>AVR ONE! 1>Target Consistency
Check>Verify All

Project>Options>Debugger>CCR>CCR>Target Consistency Check>Verify All

Project>Options>Debugger>ICE200>ICE200>Target Consistency Check>Verify
All

Project>Options>Debugger>JTAGICE>JTAGICE 1>Target Consistency
Check>Verify All

Project>Options>Debugger>JTAGICE3>JTAGICE3 1>Target Consistency
Check>Verify All

Project>Options>Debugger>JTAGICE mkII>JTAGICE mkII 1>Target
Consistency Check>Verify All

Project>Options>Debugger>Dragon>Dragon 1>Target Consistency
Check>Verify All
```

--eeprom_size

Syntax `--eeprom_size=size`

Parameters *size* The size of the built-in EEPROM area in bytes.

Applicability 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`

Applicability 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

--ice200_restore_EEPROM

Syntax	--ice200_restore_EEPROM
Applicability	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
Applicability	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= <i>speed</i>
Parameters	<i>speed</i> The JTAG clock frequency in Hz. Possible values are 0-3570000 Hz.
Applicability	The C-SPY JTAGICE, JTAGICE mkII, and AVR Dragon drivers.
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
Applicability	The C-SPY AVR ONE!, JTAGICE, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.
Description	Use this option to make the hardware reset every time the debugger is reset.



Project>Options>Debugger>Driver>Driver 2>Hardware reset on C-SPY reset

--jtagice_leave_timers_running

Syntax	--jtagice_leave_timers_running
Applicability	The C-SPY AVR ONE!, JTAGICE, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.
Description	Use this option to ensure that the timers always run, even if the application is stopped.



Project>Options>Debugger>Driver>Driver 2>Run timers in stopped mode

--jtagice_preserve_eeprom

Syntax	--jtagice_preserve_eeprom
Applicability	The C-SPY AVR ONE!, JTAGICE, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.
Description	Use this option to preserve the EEPROM contents even if device is reprogrammed.



Project>Options>Debugger>Driver>Driver 2>Preserve EEPROM contents even if device is reprogrammed

--jtagice_restore_fuse

Syntax	--jtagice_restore_fuse
Applicability	The C-SPY AVR ONE!, JTAGICE, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.

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.



Project>Options>Debugger>Driver>Driver 2>Restore fuses when ending debug session

--jtagicemkll_use_software_breakpoints

Syntax `--jtagicemkll_use_software_breakpoints`

Applicability The C-SPY AVR ONE!, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.

Description Use this option to make software breakpoints available.



Project>Options>Debugger>Driver>Driver 2>Enable software breakpoints

--macro

Syntax `--macro filename`

Parameters `filename` The C-SPY macro file to be used (filename extension mac).

Applicability Sent to cspybat.

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 *Briefly about using C-SPY macros*, page 174.

-p

Syntax `-p filename`

Parameters `filename` The device description file to be used.

Applicability All C-SPY drivers.

Description Use this option to specify the device description file to be used.

See also *Selecting a device description file*, page 49.

--plugin

Syntax `--plugin filename`

Parameters `filename` The plugin file to be used (filename extension `dll`).

Applicability Sent to `cspybat`.

Description Certain C/C++ standard library functions, for example `printf`, can be supported by C-SPY—for example, the C-SPY Terminal I/O window—instead of by real hardware devices. To enable such support in `cspybat`, a dedicated plugin module called `avrLibSupport.dll` or `avrLibSupportbat.dll` located in the `avr\bin` 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.

Note: You can use this option to include also other plugin modules, but in that case the module must be able to work with `cspybat` specifically. This means that the C-SPY plugin modules located in the `common\plugin` directory cannot normally be used with `cspybat`.

--silent

Syntax `--silent`



Applicability Sent to `cspybat`.

Description Use this option to omit the sign-on message.

--timeout

Syntax `--timeout milliseconds`

Parameters `milliseconds` The number of milliseconds before the execution stops.

Applicability	Sent to <code>cspybat</code> .
Description	Use this option to limit the maximum allowed execution time.
	 This option is not available in the IDE.
-v	
Syntax	<code>-v {0 1 2 3 4 5 6}</code>
Parameters	<div><div>0</div><div>A maximum of 256 bytes data and 8 Kbytes code. Default memory model: Tiny.</div></div> <div><div>1</div><div>A maximum of 64 Kbytes data and 8 Kbytes code. Default memory model: Tiny.</div></div> <div><div>2</div><div>A maximum of 256 bytes data and 128 Kbytes code. Default memory model: Tiny.</div></div> <div><div>3</div><div>A maximum of 64 Kbytes data and 128 Kbytes code. Default memory model: Tiny.</div></div> <div><div>4</div><div>A maximum of 16 Mbytes data and 128 Kbytes code. Default memory model: Small.</div></div> <div><div>5</div><div>A maximum of 64 Kbytes data and 8 Mbytes code. Default memory model: Small.</div></div> <div><div>6</div><div>A maximum of 16 Mbytes data and 8 Mbytes code. Default memory model: Small.</div></div>
Applicability	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

Debugger options

This chapter describes the C-SPY® options available in the IAR Embedded Workbench® IDE. More specifically, this means:

- Setting debugger options
- Reference information on debugger options
- Reference information on C-SPY driver options.

Setting debugger options

Before you start the C-SPY debugger you must set some options—both C-SPY generic options and options required for the target system (C-SPY driver-specific options). This section gives detailed information about the options in the **Debugger** category.

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:

- *Setup*, page 239
- *Images*, page 241
- *Plugins*, page 242.

- 3 On the **Setup** page, 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 sets of option pages appear.

C-SPY driver	Available options pages
C-SPY AVR ONE! driver	AVR ONE! 1, page 243 AVR ONE! 2, page 245 Communication, page 246 Extra Options, page 247

Table 37: Options specific to the C-SPY drivers you are using

C-SPY driver	Available options pages
C-SPY CCR driver	CCR, page 248 Serial Port, page 249 Extra Options, page 247
C-SPY ICE200 driver	ICE200, page 251 Serial Port, page 249 Extra Options, page 247
C-SPY JTAGICE driver	JTAGICE 1, page 253 JTAGICE 2, page 255 Serial Port, page 249 Extra Options, page 247
C-SPY JTAGICE3 driver	JTAGICE3 1, page 256 JTAGICE3 2, page 258 Communication, page 246 Extra Options, page 247
C-SPY JTAGICE mk II driver	JTAGICE mkII 1, page 259 JTAGICE mkII 2, page 262 Serial Port, page 249 Extra Options, page 247
C-SPY Dragon driver	Dragon 1, page 263 Dragon 2, page 265 Communication, page 246 Extra Options, page 247
C-SPY Simulator	Extra Options, page 247
Third-party driver	Third-Party Driver options, page 266 Extra Options, page 247

Table 37: 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

This section gives reference information on C-SPY debugger options.

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.

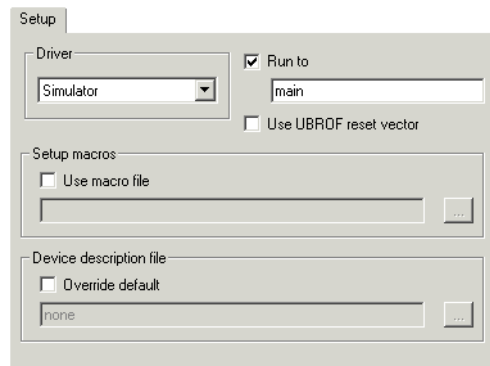


Figure 81: Debugger setup options

Driver

Selects the C-SPY driver for the target system you have:

AVR ONE!

CCR

Dragon

ICE200

JTAGICE

JTAGICE mkII

JTAGICE3

Simulator

Third-Party Driver

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 52.

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.

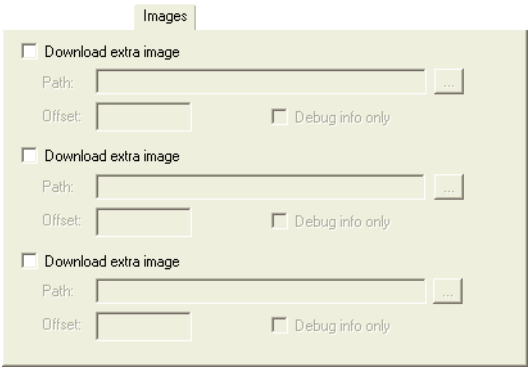


Figure 82: Debugger images options

Use 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 193.

For more information, see *Loading multiple images*, page 51.

Plugins

The **Plugins** options select the C-SPY plugin modules to be loaded and made available during debug sessions.

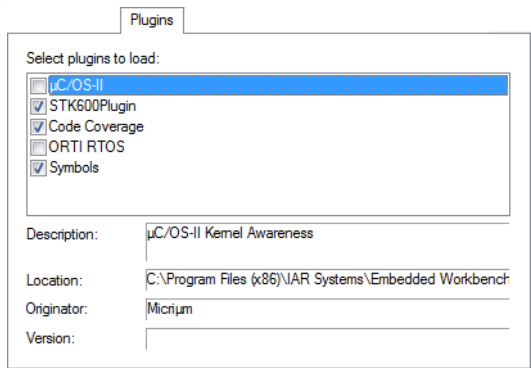


Figure 83: Debugger plugin options

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.

Reference information on C-SPY driver options

This section gives reference information on C-SPY driver options.

AVR ONE! 1

The AVR ONE! 1 options control the C-SPY driver for AVR ONE!.

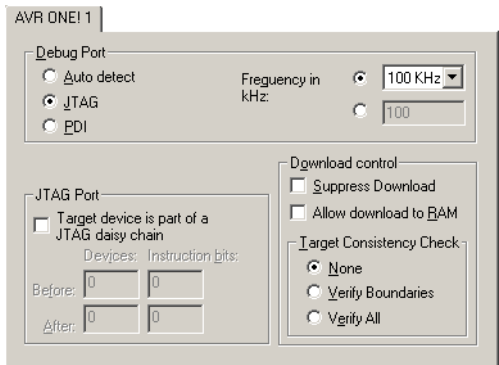


Figure 84: AVR ONE! 1 options

Debug Port

Selects the communication type. Choose between

- Auto detect

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.
- 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.
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	<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 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.</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.

AVR ONE! 2

The **AVR ONE! 2** options control the C-SPY driver for AVR ONE!.



Figure 85: AVR ONE! 2 options

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 *Using breakpoints*, page 95.

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 99.

Communication

The **Communication** options control the C-SPY driver for AVR ONE!, JTAGICE3, or Dragon.

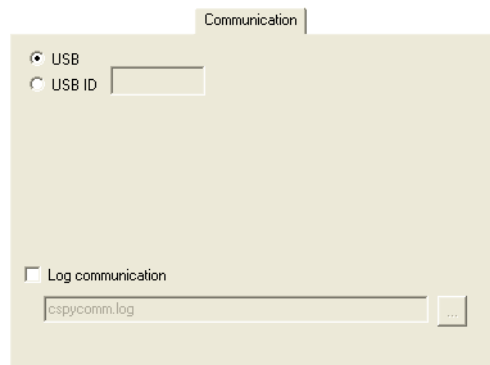


Figure 86: Communication options

These options are available:

USB	Specifies single emulator mode. Use this option for the USB port and if you have one device connected to your host computer.
USB ID	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 ONE00737 , 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 <code>cspycomm.log</code> 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.

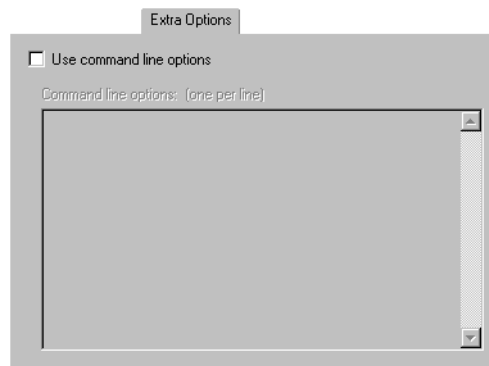


Figure 87: C-SPY driver extra options

Use command line options

Specify additional command line arguments to be passed to C-SPY (not supported by the GUI).

CCR

The CCR options control the C-SPY driver for CCR.

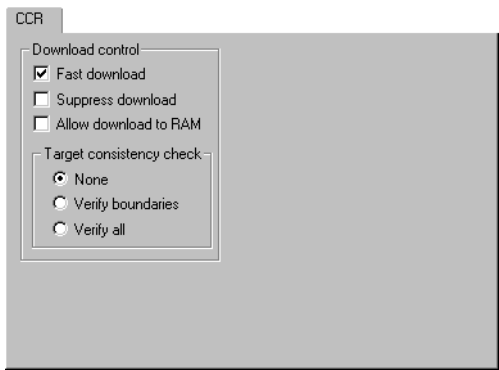


Figure 88: CCR options

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 285.

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.

Serial Port

The **Serial Port** options determine how the serial port should be used.

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.

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

Figure 89: Serial port options

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.

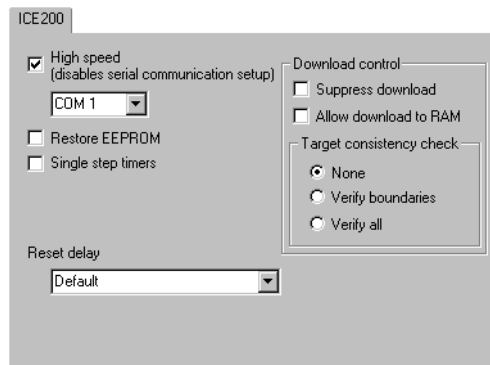


Figure 90: ICE200 options

Note that you can make temporary changes to the ICE200 options during debug sessions; see *ICE200 Options dialog box*, page 282.

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 1

The **JTAGICE 1** options control the C-SPY JTAGICE driver.

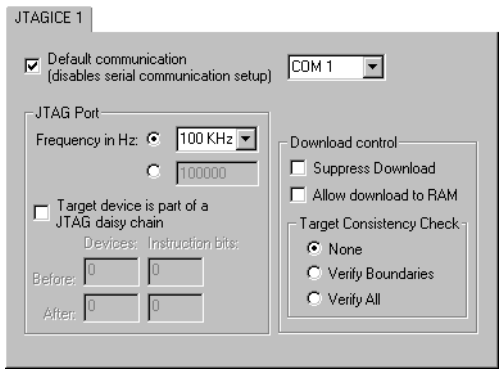


Figure 91: JTAGICE 1 options

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.

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 2

The **JTAGICE 2** options control the JTAGICE driver.

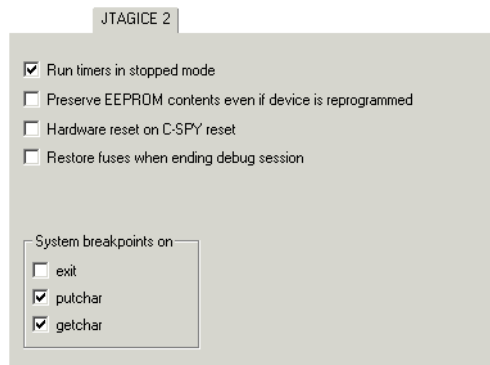


Figure 92: JTAGICE 2 options

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 99.

JTAGICE3 I

The **JTAGICE3 1** options control the C-SPY driver for JTAGICE3.

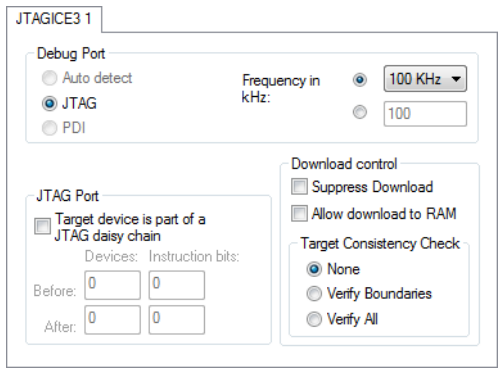


Figure 93: JTAGICE3 1 options

Debug Port

Selects the communication type. Choose between

- Auto detect

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.
- 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.
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	<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 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.</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.

JTAGICE3 2

The **JTAGICE3 2** options control the C-SPY driver for JTAGICE3.

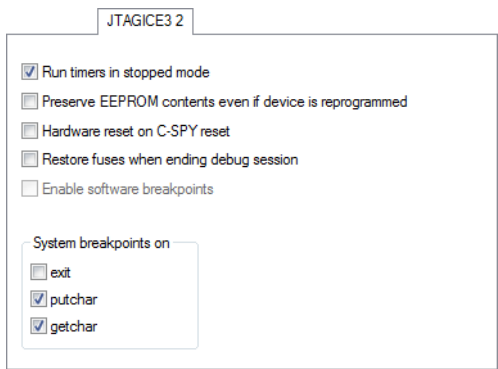


Figure 94: JTAGICE3 2 options

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 *Using breakpoints*, page 95.

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 99.

JTAGICE mkII I

The **JTAGICE mkII 1** options control the C-SPY drivers for JTAGICE mkII and Dragon.

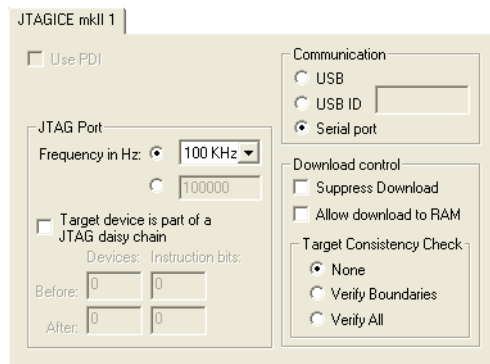


Figure 95: JTAGICE mkII 1 options

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	<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>
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.

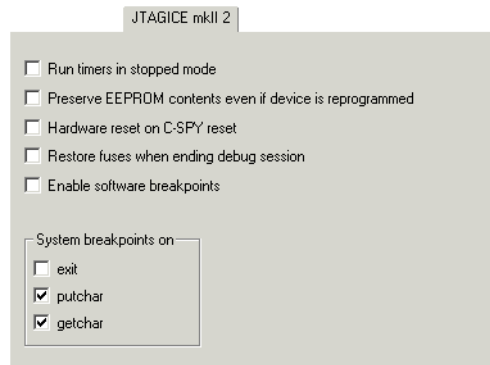


Figure 96: JTAGICE mkII 2 options

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 *Using breakpoints*, page 95.

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 99.

Dragon I

The **Dragon 1** options control the C-SPY drivers for Dragon.

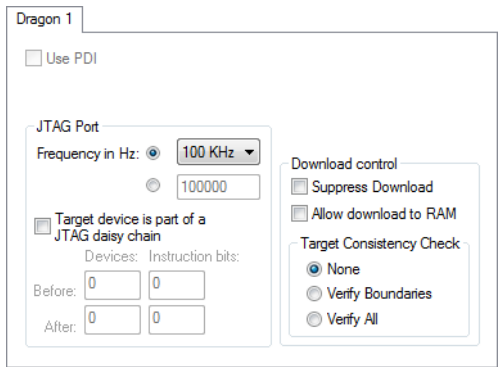


Figure 97: Dragon 1 options

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.
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	<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 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.</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	<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>None, target consistency check is not performed.</p> <p>Verify Boundaries, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.</p> <p>Verify All, 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.

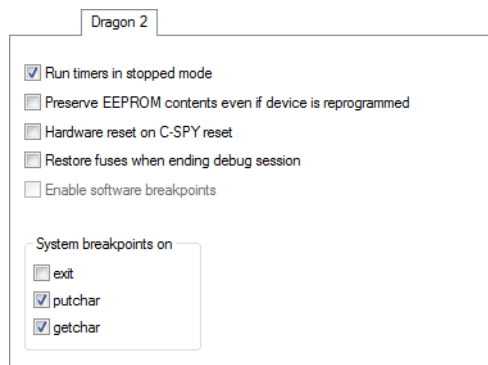


Figure 98: Dragon 2 options

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 *Using breakpoints*, page 95.

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 99.

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.

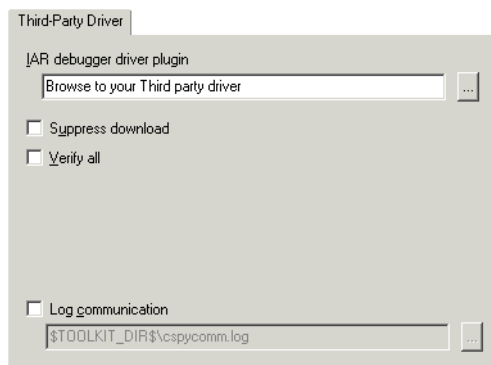


Figure 99: C-SPY Third-Party Driver options

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. More specifically, this means:

- Reference information on the C-SPY simulator
- Reference information on the C-SPY JTAGICE driver
- Reference information on the C-SPY JTAGICE mkII/Dragon driver
- Reference information on the C-SPY JTAGICE3 driver
- Reference information on the C-SPY AVR ONE! driver
- Reference information on the C-SPY ICE200 driver
- Reference information on the CCR driver.

Reference information on the C-SPY simulator

This section gives additional reference information the C-SPY simulator, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *Simulator menu*, page 270

Simulator menu

When you use the simulator driver, the **Simulator** menu is added to the menu bar.

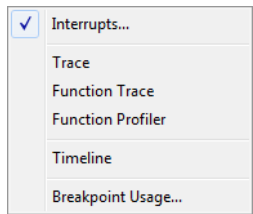


Figure 100: Simulator menu

These commands are available on the menu:

Interrupts	Displays a dialog box where you can configure C-SPY interrupt simulation, see <i>Interrupts dialog box</i> , page 170.
Trace	Opens a window which displays the collected trace data, see <i>Trace window</i> , page 143.
Function Trace	Opens a window which displays the trace data for function calls and function returns, see <i>Function Trace window</i> , page 144.
Function Profiler	Opens a window which shows timing information for the functions, see <i>Function Profiler window</i> , page 157.
Timeline	Opens a window which shows trace data for the call stack, see <i>Timeline window</i> , page 145.
Breakpoint Usage	Displays a dialog box which lists all active breakpoints, see <i>Breakpoint Usage dialog box</i> , page 108.

Reference information on the C-SPY JTAGICE driver

This section gives additional reference information on the C-SPY JTAGICE driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *JTAGICE menu*, page 271

JTAGICE menu

When you are using the C-SPY JTAGICE driver, the **JTAGICE** menu is added to the menu bar.

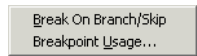


Figure 101: The JTAGICE menu

These commands are available on the menu:

- Break On Branch/Skip** Stops execution just after each branch instruction.
- Breakpoint Usage** Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 108.

Reference information on the C-SPY JTAGICE mkII/Dragon driver

This section gives additional reference information the C-SPY JTAGICE mkII driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *JTAGICE mkII menu*, page 271
- *Dragon menu*, page 272
- *Fuse Handler dialog box*, page 273.

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:

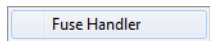


Figure 102: The JTAGICE mkII menu

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 273.

When the debugger is running, the menu looks like this:

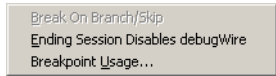


Figure 103: The JTAGICE mkII menu when the debugger is running

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** Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 108.

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:

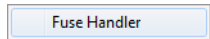


Figure 104: The Dragon menu

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 273.

When the debugger is running, the menu looks like this:

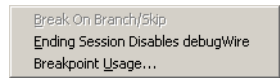


Figure 105: The Dragon menu when the debugger is running

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

Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 108.

Fuse Handler dialog box

The **Fuse Handler** dialog box is available from the **JTAGICE mkII** menu or the **Dragon** menu, respectively.

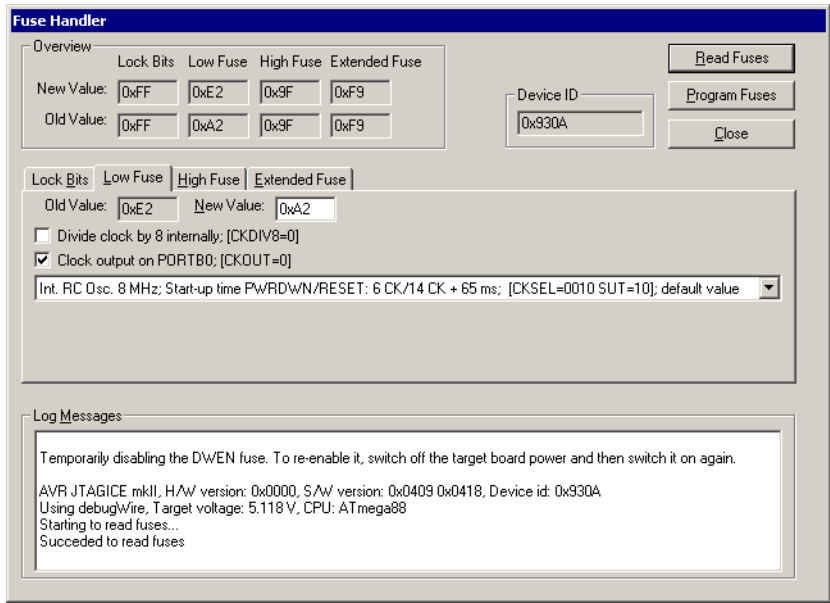


Figure 106: The JTAGICE mkII/Dragon Fuse Handler dialog box

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.

Overview

Displays an overview of the fuse settings for each fuse group.

New Value	Displays the value of the fuses reflecting the user-defined settings. In other words, the value of the fuses after they have been programmed.
Old Value	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.

Old Value	Displays the last read value of the on-chip fuses on the device.
New Value	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.

Reference information on the C-SPY JTAGICE3 driver

This section gives additional reference information on the C-SPY JTAGICE3 driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *JTAGICE3 menu*, page 275

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:

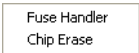


Figure 107: The JTAGICE3 menu

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 <i>Fuse Handler dialog box</i> , page 276. |
| 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:



Figure 108: The JTAGICE3 menu when the debugger is running

This command is available on the menu:

Breakpoint Usage Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 108.

Fuse Handler dialog box

The **Fuse Handler** dialog box is available from the **JTAGICE3** menu.

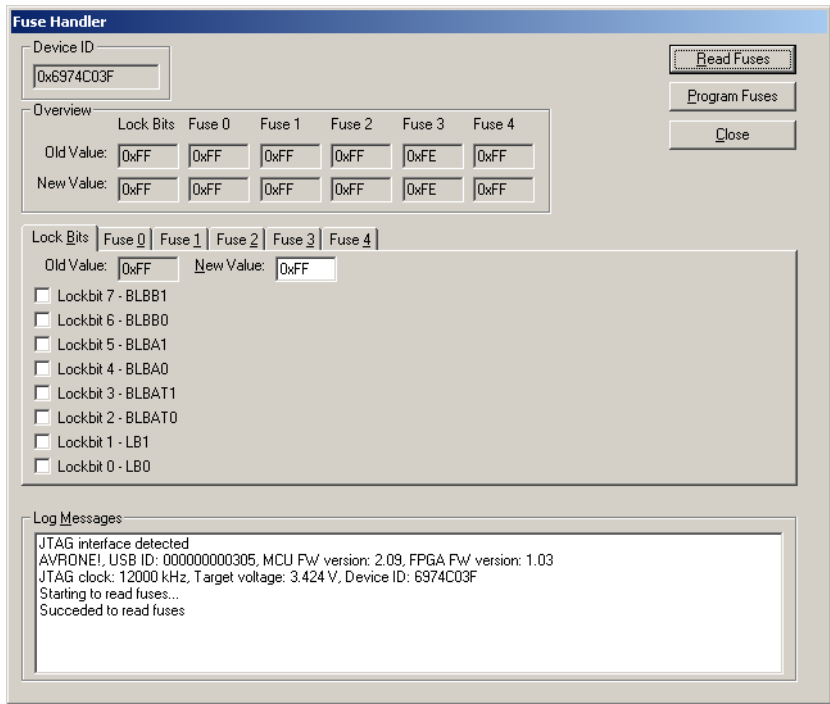


Figure 109: The JTAGICE3 Fuse Handler dialog box

Note: To use the fuse handler, the JTAG Enable fuse must be enabled on one of the pages or you can use PDI, see *PDI*, page 256. The JTAGICE3 debugger driver will guide you through this.

The fuse handler provides programming possibilities of the device-specific on-chip fuses and lock bits via JTAGICE3.

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.

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.

New Value	Displays the value of the fuses reflecting the user-defined settings. In other words, the value of the fuses after they have been programmed.
Old Value	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.

Old Value	Displays the last read value of the on-chip fuses on the device.
New Value	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.

Reference information on the C-SPY AVR ONE! driver

This section gives additional reference information on the C-SPY AVR ONE! driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *AVR ONE! menu*, page 278

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:

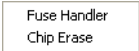


Figure 110: The AVR ONE! menu

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 <i>Fuse Handler dialog box</i> , page 279.
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:

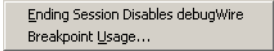


Figure 111: The AVR ONE! menu when the debugger is running

This command is available on the menu:

- Ending Session Disables debugWire**
- Disables the use of debugWIRE before ending the debug session.
- Breakpoint Usage**
- Displays a dialog box which lists all active breakpoints, see *Breakpoint Usage dialog box*, page 108.

Fuse Handler dialog box

The **Fuse Handler** dialog box is available from the **AVR ONE!** menu.

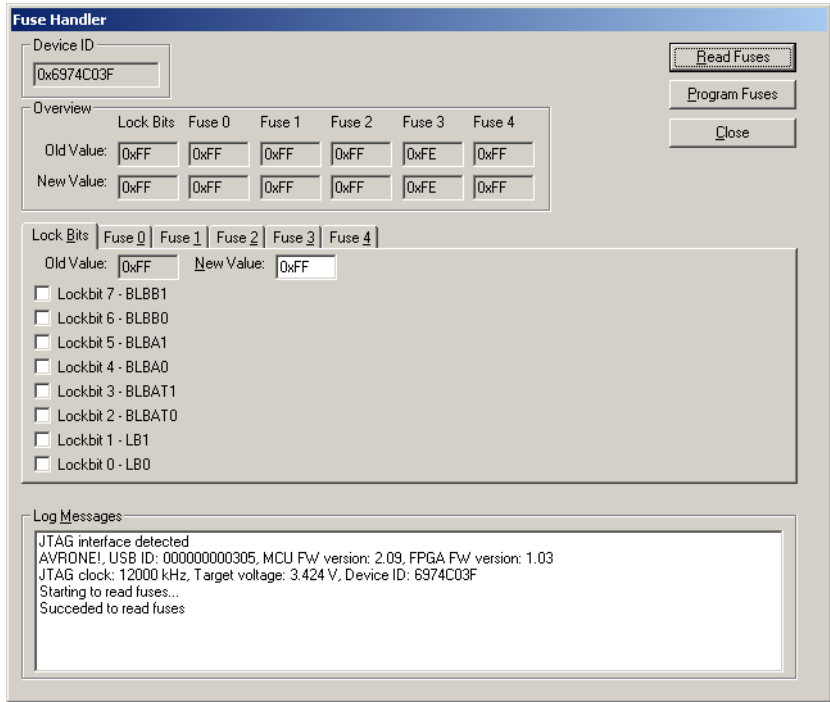


Figure 112: The AVR ONE! Fuse Handler dialog box

Note: To use the fuse handler, the JTAG Enable fuse must be enabled on one of the tabs or you can use PDI, see *PDI*, page 243. The AVR ONE! debugger driver will guide you through this.

The fuse handler provides programming possibilities of the device-specific on-chip fuses and lock bits via AVR ONE!.

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.

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.

New Value	Displays the value of the fuses reflecting the user-defined settings. In other words, the value of the fuses after they have been programmed.
Old Value	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.

Old Value	Displays the last read value of the on-chip fuses on the device.
New Value	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.

Reference information on the C-SPY ICE200 driver

This section gives additional reference information on the C-SPY ICE200 driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *ICE200 menu*, page 281
- *ICE200 Options dialog box*, page 282.

ICE200 menu

When you are using the C-SPY ICE200 driver, the **ICE200** menu is added to the menu bar.

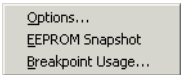


Figure 113: The ICE200 menu

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 <i>ICE200 Options dialog box</i> , page 282.
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 <i>Restore EEPROM</i> , page 251.
Breakpoint Usage	Displays a dialog box which lists all active breakpoints, see <i>Breakpoint Usage dialog box</i> , page 108.

ICE200 Options dialog box

The **ICE200 Options** dialog box is available from the **ICE200** menu.

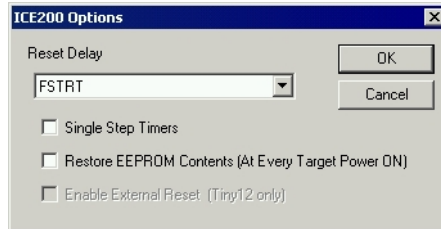


Figure 114: The ICE200 Options dialog box

Use this dialog box to make temporary changes to the ICE200 options during debug sessions.

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.

Reference information on the CCR driver

This section gives additional reference information on the C-SPY ICE200 driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *CCR menu*, page 284
- *Resolving problems*, page 285
- *Using C-SPY macros for transparent commands*, page 285.

CCR menu

When you are using the C-SPY CCR driver, the **CCR** menu is added to the menu bar.



Figure 115: The CCR menu

This command is available on the menu:

Breakpoint Usage	Displays a dialog box which lists all active breakpoints, see <i>Breakpoint Usage dialog box</i> , page 108.
-------------------------	--

RESOLVING PROBLEMS

This section includes suggestions for resolving the most common problems that can occur when debugging with the C-SPY in conjunction with the Crypto Controller ROM-monitor.

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.

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, see *Serial Port*, page 249.

Examine the linker configuration file to make sure that the application has not been linked to the wrong address.

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:

- 1** 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:

```
__transparent("AT90SC3232")
```

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.

A

A access (Complex breakpoints option)	116
Abort (Report Assert option)	78
absolute location, specifying for a breakpoint	118
Access Type (Data breakpoints option)	113
Access Type (Immediate breakpoints option)	114
Action (Code breakpoints option)	110
Action (Data breakpoints option)	113, 117
Action (Immediate breakpoints option)	115
Add to Watch Window (Symbolic Memory window context menu)	133
Add (Watch window context menu)	88
Address Range (Find in Trace option)	152
Allow download to RAM (AVR ONE! option)	244
Allow download to RAM (CCR option)	248
Allow download to RAM (Dragon option)	264
Allow download to RAM (ICE200 option)	252
Allow download to RAM (JTAGICE mkII option)	261
Allow download to RAM (JTAGICE option)	254
Allow download to RAM (JTAGICE3 option)	257
Ambiguous symbol (Resolve Symbol Ambiguity option)	94
application, built outside the IDE	50
assembler labels, viewing	85
assembler source code, fine-tuning	153
assembler symbols, using in C-SPY expressions	83
assembler variables, viewing	85
assumptions, programming experience	21
Auto Scroll (Timeline window context menu)	147
Auto window	86
Autostep settings dialog box	78
Autostep (Debug menu)	58
AVR ONE! options	243, 245
AVR ONE! (C-SPY driver)	41
hardware installation	42
menu	278
AVR ONE! (debugger option)	239
--avrone_jtag_clock (C-SPY command line option)	223

B

B access (Complex breakpoints option)	116
--backend (C-SPY command line option)	224
backtrace information	
generated by compiler	67
viewing in Call Stack window	73
batch mode, using C-SPY in	219
Baud (debugger option)	250
Big Endian (Memory window context menu)	127
blocks, in C-SPY macros	184
bold style, in this guide	25
Break At (Code breakpoints option)	109
Break At (Data breakpoints option)	113
Break At (Immediate breakpoints option)	114
Break At (Log breakpoints option)	111
Break On Branch/Skip (Dragon menu)	272
Break On Branch/Skip (JTAGICE menu)	271
Break On Branch/Skip (JTAGICE mkII menu)	272
Break (Debug menu)	57
breakpoint condition, example	104–105
Breakpoint control (Complex breakpoints option)	116
Breakpoint Usage dialog box	108
Breakpoint Usage (AVR ONE! menu)	279
Breakpoint Usage (CCR menu)	284
Breakpoint Usage (Dragon menu)	273
Breakpoint Usage (ICE200 menu)	281
Breakpoint Usage (JTAGICE menu)	271
Breakpoint Usage (JTAGICE mkII menu)	272
Breakpoint Usage (JTAGICE3 menu)	276
Breakpoint Usage (Simulator menu)	270
breakpoints	
code, example	202
complex	115
example	205
connecting a C-SPY macro	180
consumers of	99
data	112
example	207

description of	96
disabling used by Stack window	100
icons for in the IDE	97
in Memory window	103
listing all	108
log, example.	208
profiling source	154
reasons for using	95
setting	
in memory window	103
using system macros	103
using the dialog box	101
single-stepping if not available.	48
toggling	101
trace start, example	211
trace stop, example	212
types of	96
useful tips.	104
Breakpoints dialog box	
Code	109
Complex	115
Data	112
Immediate	114
Log	111
Trace Start	148
Trace Stop	149
Breakpoints window	106
Browse (Trace toolbar)	143

C

C function information, in C-SPY.	67
C symbols, using in C-SPY expressions	83
C variables, using in C-SPY expressions	82
call chain, displaying in C-SPY	67
Call stack information.	67
Call Stack window	73
for backtrace information.	67
Call Stack (Timeline window context menu)	147

__cancelAllInterrupts (C-SPY system macro)	189
__cancelInterrupt (C-SPY system macro).	189
CCR options	248
CCR (C-SPY driver).	45
menu	284
CCR (debugger option).	239
Chip Erase (JTAGICE3 menu)	275
Clear All (Debug Log window context menu)	77
Clear trace data (Trace toolbar).	143
__clearBreak (C-SPY system macro)	189
CLIB, documentation	23
__closeFile (C-SPY system macro)	190
code breakpoints, overview.	96
Code Coverage window	162
Code Coverage (Disassembly window context menu)	71
--code_coverage_file (C-SPY command line option)	224
code, covering execution of	162
command line options.	223
typographic convention	25
command prompt icon, in this guide.	25
Communication (JTAGICE mkII option)	260
complex breakpoints, overview.	97
Complex data (Complex breakpoints option)	116
computer style, typographic convention	25
conditional statements, in C-SPY macros	183
Conditions (Code breakpoints option)	110
Conditions (Data breakpoints option)	113
Conditions (Log breakpoints option)	112
context menu, in windows.	82
conventions, used in this guide	24
Copy Window Contents (Disassembly window context menu)	72
Copy (Debug Log window context menu)	77
copyright notice	2
--cpu (C-SPY command line option).	224
cspybat	219
current position, in C-SPY Disassembly window	70
cursor, in C-SPY Disassembly window.	70
--cycles (C-SPY command line option)	225

- C-SPY
 - batch mode, using in 219
 - debugger systems, overview of 31
 - differences between drivers 33
 - environment overview 27
 - plugin modules, loading 49
 - setting up 47–48
 - starting the debugger 49
 - C-SPY drivers
 - AVR ONE! 41
 - CCR 45
 - Dragon 37
 - ICE200 43
 - JTAGICE 35
 - JTAGICE mkII 37
 - JTAGICE3 39
 - overview 33
 - specifying 239
 - C-SPY expressions 82
 - evaluating 92
 - in C-SPY macros 183
 - Tooltip watch, using 81
 - Watch window, using 81
 - C-SPY macro "__message" style (Log breakpoints option) 111
 - C-SPY macros
 - blocks 184
 - conditional statements 183
 - C-SPY expressions 183
 - dialog box, using 177
 - examples 175
 - checking status of register 179
 - checking the status of WDT 179
 - creating a log macro 180
 - executing 175
 - connecting to a breakpoint 180
 - using Quick Watch 179
 - using setup macro and setup file 178
 - functions 83, 181
 - loop statements 183
 - macro statements 183
 - setup macro file 174
 - executing 178
 - setup macro functions 174
 - summary 186
 - system macros, summary of 187
 - using 173
 - variables 84, 182
 - C-SPY options 237
 - Extra Options 247
 - Images 241
 - Plugins 242
 - Setup 239
 - C-SPYLink 33
 - C++ Exceptions
 - Break on Throw (Debug menu) 58
 - Break on Uncaught Exception (Debug menu) 58
 - C++ terminology 24
- ## D
- d (C-SPY command line option) 225
 - Data bits (debugger option) 250
 - data breakpoints, overview 97
 - Data Coverage (Memory window context menu) 127
 - data coverage, in Memory window 126
 - ddf (filename extension), selecting a file 49
 - Debug Log window 76
 - Debug Log window context menu 76
 - Debug menu (C-SPY main window) 57
 - Debug Port (AVR ONE! option) 243
 - Debug Port (JTAGICE3 option) 256
 - Debug (Report Assert option) 78
 - debugger concepts, definitions of 30
 - debugger drivers. *See* C-SPY drivers. 35, 37, 39, 41, 43, 45
 - debugger system overview 31
 - debugging projects
 - externally built applications 50
 - loading multiple images 51

debugging, RTOS awareness	29
debugWIRE	272–273
Default communication (JTAGICE option)	253
Default communication (Serial Port option)	249
__delay (C-SPY system macro)	190
Delay (Autostep Settings option)	79
Delete (Breakpoints window context menu)	107
Device description file (debugger option)	240
device description files	49
definition of	52
memory zones	123
modifying	52
register zone	123
Device ID (Fuse Handler option)	274, 277, 280
Devices After (AVR ONE! option)	244
Devices After (Dragon option)	264
Devices After (JTAGICE mkII option)	260
Devices After (JTAGICE option)	254
Devices After (JTAGICE3 option)	257
Devices Before (AVR ONE! option)	244
Devices Before (Dragon option)	264
Devices Before (JTAGICE mkII option)	260
Devices Before (JTAGICE option)	254
Devices Before (JTAGICE3 option)	257
Disable All (Breakpoints window context menu)	107
Disable (Breakpoints window context menu)	107
__disableInterrupts (C-SPY system macro)	190
--disable_internal_eeprom (C-SPY command line option)	226
--disable_interrupts (C-SPY command line option)	226
Disassembly window	69
context menu	71
disclaimer	2
DLIB, documentation	23
do (macro statement)	184
document conventions	24
documentation	
overview of guides	23
overview of this guide	22
this guide	21

Download control (AVR ONE! option)	244
Download control (CCR option)	248
Download control (Dragon option)	264
Download control (ICE200 option)	252
Download control (JTAGICE mkII option)	261
Download control (JTAGICE option)	254
Download control (JTAGICE3 option)	257
--download_only (C-SPY command line option)	226
Dragon options	263, 265
Dragon (C-SPY driver)	37
menu	272
Dragon (debugger option)	239
Driver (debugger option)	239
__driverType (C-SPY system macro)	191
--drv_communication (C-SPY command line option)	226
--drv_communication_log (C-SPY command line option)	227
--drv_debug_port (C-SPY command line option)	227
--drv_download_data (C-SPY command line option)	228
--drv_dragon (C-SPY command line option)	228
--drv_set_exit_breakpoint (C-SPY command line option)	229
--drv_set_getchar_breakpoint (C-SPY command line option)	229
--drv_set_putchar_breakpoint (C-SPY command line option)	229
--drv_suppress_download (C-SPY command line option)	230
--drv_use_PDI (C-SPY command line option)	230
--drv_verify_download (C-SPY command line option)	230

E

Edit Expressions (Trace toolbar)	144
Edit Settings (Trace toolbar)	143
Edit (Breakpoints window context menu)	107
edition, of this guide	2
EEPROM	
contents, preserving in AVR ONE!	245
contents, preserving in Dragon	265
contents, preserving in JTAGICE	255
contents, preserving in JTAGICE mkII	262

- contents, preserving in JTAGICE3 258
- eeprom_size (C-SPY command line option) 231
- Embedded C++ Technical Committee 24
- Enable All (Breakpoints window context menu). 107
- Enable External Reset (ICE200 Options option) 284
- Enable interrupt simulation (Interrupt Setup option). 170
- Enable Log File (Log File option). 77
- Enable software breakpoints (AVR ONE! option). 246
- Enable software breakpoints (Dragon option). 265
- Enable software breakpoints (JTAGICE mkII option). 262
- Enable software breakpoints (JTAGICE3 option). 259
- Enable (Breakpoints window context menu). 107
- Enable (Timeline window context menu) 147
- __enableInterrupts (C-SPY system macro). 191
- Enable/Disable Breakpoint (Call Stack window context menu) 74
- Enable/Disable Breakpoint (Disassembly window context menu) 72
- Enable/Disable (Trace toolbar) 143
- Ending Session Disables debugWire (AVR ONE! menu) . 279
- Ending Session Disables debugWIRE (Dragon menu) . . 273
- Ending Session Disables debugWIRE (JTAGICE mkII menu). 272
- enhanced_core (C-SPY command line option). 231
- Enter Location dialog box. 118
- __evaluate (C-SPY system macro) 192
- examples
 - C-SPY macros 175
 - interrupts, timer 168
 - macros
 - checking status of register. 179
 - checking status of WDT 179
 - creating a log macro 180
 - using Quick Watch 179
 - performing tasks and continue execution 105
 - tracing incorrect function arguments 104
- execUserExit (C-SPY setup macro) 186
- execUserPreload (C-SPY setup macro). 186
- execUserPreReset (C-SPY setup macro). 186
- execUserReset (C-SPY setup macro) 186

- execUserSetup (C-SPY setup macro) 186
- executed code, covering 162
- execution history, tracing 142
- expressions. *See* C-SPY expressions
- Extra Options, for C-SPY 247

F

- File format (Memory Save option) 129
- file types
 - device description, specifying in IDE 49
 - macro 49, 240
- filename extensions
 - ddf, selecting device description file 49
 - mac, using macro file. 49
- Filename (Memory Restore option) 129
- Filename (Memory Save option). 129
- Fill dialog box. 130
- Find in Trace dialog box. 151
- Find in Trace window 152
- Find (Memory window context menu) 128
- Find (Trace toolbar) 143
- first activation time (interrupt property)
 - definition of 166
- flash memory, load library module to 193
- for (macro statement) 183
- formats, C-SPY input 29
- Frequency in Hz (AVR ONE! option). 243
- Frequency in Hz (JTAGICE mkII option). 260, 263
- Frequency in Hz (JTAGICE option) 253
- Frequency in Hz (JTAGICE3 option) 256
- Function Profiler window 157
- Function Profiler (Simulator menu) 270
- Function Trace window. 144
- Function Trace (Simulator menu) 270
- functions, C-SPY running to when starting. 48, 240
- Fuse Handler dialog box. 273, 276, 279
- Fuse Handler (AVR ONE! menu) 278
- Fuse Handler (Dragon menu) 272

Fuse Handler (JTAGICE mkII menu)	271
Fuse Handler (JTAGICE3 menu)	275

G

__getCycleCounter (C-SPY system macro)	192
Go to Source (Breakpoints window context menu)	107
Go to Source (Call Stack window context menu)	74
Go To Source (Timeline window context menu)	147
Go (Debug menu)	57, 66

H

Handshaking (debugger option)	250
Hardware reset on C-SPY reset (AVR ONE! option)	245
Hardware reset on C-SPY reset (Dragon option)	265
Hardware reset on C-SPY reset (JTAGICE mkII option)	262
Hardware reset on C-SPY reset (JTAGICE option)	255
Hardware reset on C-SPY reset (JTAGICE3 option)	258
High speed (ICE200 option)	251
highlighting, in C-SPY	66
hold time (interrupt property), definition of	167

I

IAR debugger driver plugin (debugger option)	266
ICE200 options	251
ICE200 Options dialog box	282
ICE200 (C-SPY driver)	43
hardware installation	44
menu	281
ICE200 (debugger option)	239
--ice200_restore_EEPROM (C-SPY command line option)	232
--ice200_single_step_timers (C-SPY command line option)	232
icons, in this guide	25
if else (macro statement)	183

if (macro statement)	183
Ignore (Report Assert option)	78
Images window	60
Images, loading multiple	241
immediate breakpoints, overview	97
Include (Log File option)	77
input formats, C-SPY	29
Input Mode dialog box	75
input, special characters in Terminal I/O window	75
installation directory	24
Instruction bits After (AVR ONE! option)	244
Instruction bits After (Dragon option)	264
Instruction bits After (JTAGICE mkII option)	261
Instruction bits After (JTAGICE option)	254
Instruction bits After (JTAGICE3 option)	257
Instruction bits Before (AVR ONE! option)	244
Instruction bits Before (Dragon option)	264
Instruction bits Before (JTAGICE mkII option)	260
Instruction bits Before (JTAGICE option)	254
Instruction bits Before (JTAGICE3 option)	257
Instruction Profiling (Disassembly window context menu)	71
Intel-extended, C-SPY input format	29
Intel-extended, C-SPY output format	32
interrupt system, using device description file	167
Interrupt (Timeline window context menu)	147
interrupts	
adapting C-SPY system for target hardware	167
simulated, introduction to	165
timer, example	168
using system macros	167
Interrupts dialog box	170
Interrupts (Simulator menu)	270
__isBatchMode (C-SPY system macro)	192
italic style, in this guide	25

J

JTAG daisy chain (AVR ONE! option)	244
JTAG daisy chain (Dragon option)	264

JTAG daisy chain (JTAGICE mkII option)	260
JTAG daisy chain (JTAGICE option)	253
JTAG daisy chain (JTAGICE3 option)	257
JTAG Port (AVR ONE! option)	244
JTAG Port (Dragon option)	263
JTAG Port (JTAGICE mkII option)	260
JTAG Port (JTAGICE option)	253
JTAG Port (JTAGICE3 option)	257
JTAGICE mkII options	259, 262
JTAGICE mkII (C-SPY driver)	37
hardware installation	38
menu	271
JTAGICE mkII (debugger option)	239
JTAGICE options	253, 255
JTAGICE (C-SPY driver)	35
hardware installation	36
menu	271
JTAGICE (debugger option)	239
--jtagicemkII_use_software_breakpoints (C-SPY command line option)	234
--jtagice_clock (C-SPY command line option)	232
--jtagice_do_hardware_reset (C-SPY command line option)	233
--jtagice_leave_timers_running (C-SPY command line option)	233
--jtagice_preserve_eeprom (C-SPY command line option)	233
--jtagice_restore_fuse (C-SPY command line option)	233
JTAGICE3 options	256, 258
JTAGICE3 (C-SPY driver)	39
hardware installation	40
menu	275
JTAGICE3 (debugger option)	239

L

labels (assembler), viewing	85
Length (Fill option)	130
lightbulb icon, in this guide	25
Little Endian (Memory window context menu)	127
__loadImage(C-SPY system macro)	193

loading multiple debug files, list currently loaded	60
loading multiple images	51
Locals window	87
log breakpoints, overview	96
Log communication (debugger option)	250, 267
Log File dialog box	77
Log Messages (Fuse Handler option)	275, 278, 281
Logging>Set Log file (Debug menu)	58
Logging>Set Terminal I/O Log file (Debug menu)	58
loop statements, in C-SPY macros	183

M

mac (filename extension), using a macro file	49
--macro (C-SPY command line option)	234
Macro Configuration dialog box	177
macro files, specifying	49, 240
macro statements	183
macros	
executing	175
using	173
Macros (Debug menu)	58
main function, C-SPY running to when starting	48, 240
Memory Fill (Memory window context menu)	128
Memory Restore dialog box	129
Memory Restore (Memory window context menu)	128
Memory Save dialog box	128
Memory Save (Memory window context menu)	128
Memory window	125
memory zones	122
in device description file	123
__memoryRestore (C-SPY system macro)	194
__memoryRestoreFromFile (C-SPY system macro)	194
__memorySave (C-SPY system macro)	195
__memorySaveToFile (C-SPY system macro)	196
Memory>Restore (Debug menu)	58
Memory>Save (Debug menu)	58
menu bar, C-SPY-specific	56
Message (Log breakpoints option)	111

migration, from earlier IAR compilers	23
MISRA C, documentation	23
Mixed Mode (Disassembly window context menu)	72
Motorola	
C-SPY input format	29
C-SPY output format	32
Move to PC (Disassembly window context menu)	71

N

naming conventions	25
Navigate (Timeline window context menu)	146
New Breakpoint (Breakpoints window context menu)	108
Next Statement (Debug menu)	57
Next Symbol (Symbolic Memory window context menu)	133

O

__openFile (C-SPY system macro).	196
Operation (Fill option)	130
operators, sizeof in C-SPY	84
optimizations, effects on variables	84
options	
in the IDE	237
on the command line	223, 247
Options (Stack window context menu)	135
__orderInterrupt (C-SPY system macro).	198
Originator (debugger option)	242

P

-p (C-SPY command line option)	234
parameters	
tracing incorrect values of	67
typographic convention	25
Parity (debugger option)	250
part number, of this guide	2
peripheral units, device-specific	52

Please select one symbol	
(Resolve Symbol Ambiguity option)	94
--plugin (C-SPY command line option)	235
plugin modules (C-SPY).	32
loading	49
Plugins (C-SPY options).	242
Port (Serial Port option)	249
prerequisites, programming experience.	21
Preserve EEPROM contents (AVR ONE! option).	245
Preserve EEPROM contents (Dragon option).	265
Preserve EEPROM contents (JTAGICE mkII option).	262
Preserve EEPROM contents (JTAGICE option)	255
Preserve EEPROM contents (JTAGICE3 option)	258
Previous Symbol (Symbolic Memory window context menu)	133
probability (interrupt property), definition of	167
Profile Selection (Timeline window context menu)	148
profiling	
on function level	155
on instruction level.	155
profiling information, on functions and instructions	153
profiling sources	
breakpoints	154
trace (calls)	154, 158
trace (flat)	154, 158
program execution, in C-SPY	63
Program Fuses (Fuse Handler option).	275, 278, 281
programming experience.	21
projects, for debugging externally built applications.	50
publication date, of this guide.	2

Q

Quick Watch window	92
executing C-SPY macros	179

R

Read Fuses (Fuse Handler option)	274, 277, 280
--	---------------

- `__readFile` (C-SPY system macro) 198
- `__readFileByte` (C-SPY system macro) 199
- reading guidelines 21
- `__readMemoryByte` (C-SPY system macro) 199
- `__readMemory8` (C-SPY system macro) 199
- `__readMemory16` (C-SPY system macro) 200
- `__readMemory32` (C-SPY system macro) 200
- reference information, typographic convention 25
- Refresh (Debug menu) 58
- register groups 122
 - predefined, enabling 136
- Register window 136
- registered trademarks 2
- `__registerMacroFile` (C-SPY system macro) 201
- registers, displayed in Register window 136
- Remove (Watch window context menu) 88
- repeat interval (interrupt property), definition of 166
- Replace (Memory window context menu) 128
- Report Assert dialog box 78
- Reset Delay (ICE200 Options option) 282
- Reset (Debug menu) 57
- `__resetFile` (C-SPY system macro) 201
- Resolve Source Ambiguity dialog box 119
- Restore EEPROM Contents (ICE200 Options option) 283
- Restore EEPROM (ICE200 option) 251
- Restore fuses when ending debug session (AVR ONE! option) 246
- Restore fuses when ending debug session (JTAGICE option) 255
- Restore fuses when ending debug session (JTAGICE3 option) 259
- Restore (Memory Restore option) 130
- return (macro statement) 184
- ROM-monitor, definition of 32
- RTOS awareness debugging 29
- RTOS awareness (C-SPY plugin module) 30
- Run timers in stopped mode (AVR ONE! option) 245
- Run timers in stopped mode (Dragon option) 265
- Run timers in stopped mode (JTAGICE mkII option) 262
- Run timers in stopped mode (JTAGICE option) 255

- Run timers in stopped mode (JTAGICE3 option) 258
- Run to Cursor (Call Stack window context menu) 74
- Run to Cursor (Debug menu) 58
- Run to Cursor (Disassembly window context menu) 71
- Run to Cursor, command for executing 66
- Run to (C-SPY option) 48
- Run to (debugger option) 240

S

- Save (Memory Save option) 129
- Save (Trace toolbar) 143
- Select All (Debug Log window context menu) 77
- Select Graphs (Timeline window context menu) 147
- Select plugins to load (debugger option) 242
- serial port setup, hardware drivers 249
- Set Data Breakpoint (Memory window context menu) 128
- Set Next Statement (Debug menu) 58
- Set Next Statement (Disassembly window context menu) 72
- `__setCodeBreak` (C-SPY system macro) 202
- `__setComplexBreak` (C-SPY system macro) 203
- `__setDataBreak` (C-SPY system macro) 206
- `__setLogBreak` (C-SPY system macro) 207
- `__setSimBreak` (C-SPY system macro) 209
- `__setTraceStartBreak` (C-SPY system macro) 210
- `__setTraceStopBreak` (C-SPY system macro) 211
- setup macro functions 174
 - reserved names 186
- Setup macros (debugger option) 240
- Setup (C-SPY options) 239
- SFR
 - in Register window 137
 - using as assembler symbols 83
- Show all images (Images window context menu) 61
- Show Arguments (Call Stack window context menu) 74
- Show As (Watch window context menu) 88
- Show offsets (Stack window context menu) 135
- Show only (Image window context menu) 61
- Show variables (Stack window context menu) 135

--silent (C-SPY command line option)	235
simulating interrupts, enabling/disabling	170
simulator driver, selecting	34
Simulator menu.	270
Simulator (debugger option)	239
simulator, introduction	34
Single Step Timers (ICE200 Options option)	283
Single step timers (ICE200 option)	252
Size (Code breakpoints option)	109
Size (Data breakpoints option)	113
sizeof	84
__sourcePosition (C-SPY system macro)	212
special function registers (SFR)	
in Register window	137
using as assembler symbols	83
stack usage, computing	123
Stack window	133
stack.mac	174
standard C, sizeof operator in C-SPY	84
Start address (Fill option)	130
Start address (Memory Save option)	129
Statics window	89
stdin and stdout, redirecting to C-SPY window	74
Step Into (Debug menu)	57
Step Into, description	65
Step Out (Debug menu)	57
Step Out, description.	66
Step Over (Debug menu)	57
Step Over, description.	65
step points, definition of	64
Stop address (Memory Save option)	129
Stop bits (debugger option)	250
Stop Debugging (Debug menu).	57
__strFind (C-SPY system macro)	212
__subString (C-SPY system macro)	213
Suppress download	
AVR ONE! option	244
CCR option	248
debugger option	266

Dragon option	264
ICE200 option	252
JTAGICE mkII option	261
JTAGICE option	254
JTAGICE3 option	257
Symbolic Memory window.	131
Symbols window	93
symbols, using in C-SPY expressions	82
System breakpoints on (AVR ONE! option)	246
System breakpoints on (Dragon option)	265
System breakpoints on (JTAGICE mkII option)	262
System breakpoints on (JTAGICE option)	255
System breakpoints on (JTAGICE3 option)	259

T

Target Consistency Check (AVR ONE! option)	245
Target Consistency Check (CCR option)	249
Target Consistency Check (Dragon option).	264
Target Consistency Check (ICE200 option)	252
Target Consistency Check (JTAGICE mkII option)	260–261
Target Consistency Check (JTAGICE option).	254
Target Consistency Check (JTAGICE3 option).	258
Target device is part of a JTAG daisy chain (AVR ONE! option)	244
Target device is part of a JTAG daisy chain (Dragon option)	264
Target device is part of a JTAG daisy chain (JTAGICE mkII option)	260
Target device is part of a JTAG daisy chain (JTAGICE option)	253
Target device is part of a JTAG daisy chain (JTAGICE3 option)	257
target system, definition of	31
__targetDebuggerVersion (C-SPY system macro)	213
Terminal IO Log Files (Terminal IO Log Files option).	76
Terminal I/O Log Files dialog box	75
Terminal I/O window	68, 74
terminology.	24

Text search (Find in Trace option) 151

Third-Party Driver (debugger options) 266

Third-Party Driver (debugger option) 239

Time Axis Unit (Timeline window context menu) 147

Timeline window 145

Timeline (Simulator menu) 270

--timeout (C-SPY command line option) 235

timer interrupt, example 168

timers (AVR ONE!), running in stopped mode 245

timers (Dragon), running in stopped mode 265

timers (JTAGICE mkII), running in stopped mode 262

timers (JTAGICE), running in stopped mode 255

timers (JTAGICE3), running in stopped mode 258

Toggle Breakpoint (Code) (Call Stack window context menu) 74

Toggle Breakpoint (Code) (Disassembly window context menu) 72

Toggle Breakpoint (Log) (Call Stack window context menu) 74

Toggle Breakpoint (Log) (Disassembly window context menu) 72

Toggle Breakpoint (Trace Start) (Disassembly window context menu) 72

Toggle Breakpoint (Trace Stop) (Disassembly window context menu) 72

Toggle source (Trace toolbar) 143

__toLower (C-SPY system macro) 214

tools icon, in this guide 25

__toString (C-SPY system macro) 214

__toUpper (C-SPY system macro) 215

Trace Expressions window 150

trace start and stop breakpoints, overview 96

Trace Start breakpoints dialog box 148

Trace Stop breakpoints dialog box 149

Trace window 143

trace (calls), profiling source 154, 158

trace (flat), profiling source 154, 158

Trace (Simulator menu) 270

trace, Timeline window 145

trademarks 2

__transparent (C-SPY system macro) 215

typographic conventions 25

U

UBROF 29

Unavailable, C-SPY message 85

Universal Binary Relocatable Object Format. *See* UBROF

__unloadImage(C-SPY system macro) 216

USB

- AVR ONE! option 247
- JTAGICE mkII option 260

USB ID (AVR ONE! option) 247

USB ID (JTAGICE mkII option) 260

Use command line options (C-SPY driver option) 247

Use Extra Images (debugger option) 241

Use PDI (JTAGICE mkII option) 259, 263

Use UBROF reset vector (debugger option) 240

user application, definition of 31

V

-v (C-SPY command line option) 236

Value (Fill option) 130

variables

- effects of optimizations 84
- information, limitation on 84
- using in C-SPY expressions 82

variance (interrupt property), definition of 167

Verify all (debugger option) 266

version number, of this guide 2

visualSTATE, C-SPY plugin module for 33

W

warnings icon, in this guide 25

Watch window 87

- using 81

web sites, recommended 24

while (macro statement)	183
windows, specific to C-SPY	59
With I/O emulation modules (linker option), using	74
__writeFile (C-SPY system macro)	216
__writeFileByte (C-SPY system macro)	217
__writeMemoryByte (C-SPY system macro)	217
__writeMemory8 (C-SPY system macro)	217
__writeMemory16 (C-SPY system macro)	217
__writeMemory32 (C-SPY system macro)	218

Z

zone	
defined in device description file	123
in C-SPY	122
Zone (Fill option)	130
Zone (Memory Restore option)	129
Zone (Memory Save option)	129
Zone (Memory window context menu)	127
Zoom (Timeline window context menu)	147

Symbols

__cancelAllInterrupts (C-SPY system macro)	189
__cancelInterrupt (C-SPY system macro)	189
__clearBreak (C-SPY system macro)	189
__closeFile (C-SPY system macro)	190
__delay (C-SPY system macro)	190
__disableInterrupts (C-SPY system macro)	190
__driverType (C-SPY system macro)	191
__enableInterrupts (C-SPY system macro)	191
__evaluate (C-SPY system macro)	192
__fmessage (C-SPY macro statement)	184
__getCycleCounter (C-SPY system macro)	192
__isBatchMode (C-SPY system macro)	192
__loadImage (C-SPY system macro)	193
__memoryRestore (C-SPY system macro)	194
__memoryRestoreFromFile (C-SPY system macro)	194
__memorySave (C-SPY system macro)	195

__memorySaveToFile (C-SPY system macro)	196
__message (C-SPY macro statement)	184
__openFile (C-SPY system macro)	196
__orderInterrupt (C-SPY system macro)	198
__readFile (C-SPY system macro)	198
__readFileByte (C-SPY system macro)	199
__readMemoryByte (C-SPY system macro)	199
__readMemory8 (C-SPY system macro)	199
__readMemory16 (C-SPY system macro)	200
__readMemory32 (C-SPY system macro)	200
__registerMacroFile (C-SPY system macro)	201
__resetFile (C-SPY system macro)	201
__setCodeBreak (C-SPY system macro)	202
__setComplexBreak (C-SPY system macro)	203
__setDataBreak (C-SPY system macro)	206
__setLogBreak (C-SPY system macro)	207
__setSimBreak (C-SPY system macro)	209
__setTraceStartBreak (C-SPY system macro)	210
__setTraceStopBreak (C-SPY system macro)	211
__smessage (C-SPY macro statement)	184
__sourcePosition (C-SPY system macro)	212
__strFind (C-SPY system macro)	212
__subString (C-SPY system macro)	213
__targetDebuggerVersion (C-SPY system macro)	213
__toLowerCase (C-SPY system macro)	214
__toString (C-SPY system macro)	214
__toUpper (C-SPY system macro)	215
__transparent (C-SPY system macro)	215
__unloadImage (C-SPY system macro)	216
__writeFile (C-SPY system macro)	216
__writeFileByte (C-SPY system macro)	217
__writeMemoryByte (C-SPY system macro)	217
__writeMemory8 (C-SPY system macro)	217
__writeMemory16 (C-SPY system macro)	217
__writeMemory32 (C-SPY system macro)	218
-d (C-SPY command line option)	225
-p (C-SPY command line option)	234
-v (C-SPY command line option)	236
--avrone_jtag_clock (C-SPY command line option)	223

--backend (C-SPY command line option) 224
 --code_coverage_file (C-SPY command line option) 224
 --cpu (C-SPY command line option) 224
 --cycles (C-SPY command line option) 225
 --disable_internal_eeprom (C-SPY command
 line option) 226
 --disable_interrupts (C-SPY command line option) 226
 --download_only (C-SPY command line option) 226
 --drv_communication (C-SPY command line option) 226
 --drv_communication_log (C-SPY command
 line option) 227
 --drv_debug_port (C-SPY command line option) 227
 --drv_download_data (C-SPY command line option) 228
 --drv_dragon (C-SPY command line option) 228
 --drv_set_exit_breakpoint (C-SPY command
 line option) 229
 --drv_set_getchar_breakpoint
 (C-SPY command line option) 229
 --drv_set_putchar_breakpoint
 (C-SPY command line option) 229
 --drv_suppress_download (C-SPY command
 line option) 230
 --drv_use_PDI (C-SPY command line option) 230
 --drv_verify_download (C-SPY command line option) . . 230
 --eeprom_size (C-SPY command line option) 231
 --enhanced_core (C-SPY command line option) 231
 --ice200_restore_EEPROM
 (C-SPY command line option) 232
 --ice200_single_step_timers
 (C-SPY command line option) 232
 --jtagicemkII_use_software_breakpoints (C-SPY command
 line option) 234
 --jtagice_clock (C-SPY command line option) 232
 --jtagice_do_hardware_reset
 (C-SPY command line option) 233
 --jtagice_leave_timers_running
 (C-SPY command line option) 233
 --jtagice_preserve_eeprom (C-SPY command
 line option) 233
 --jtagice_restore_fuse (C-SPY command line option) . . . 233

--macro (C-SPY command line option) 234
 --plugin (C-SPY command line option) 235
 --silent (C-SPY command line option) 235
 --timeout (C-SPY command line option) 235
 --64bit_doubles (C-SPY command line option) 223
 --64k_flash (C-SPY command line option) 223

Numerics

1x Units (Memory window context menu) 127
 1x Units (Stack window context menu) 135
 1x Units (Symbolic Memory window context menu) 133
 2x Units (Memory window context menu) 127
 2x Units (Stack window context menu) 135
 2x Units (Symbolic Memory window context menu) 133
 4x Units (Memory window context menu) 127
 4x Units (Stack window context menu) 135
 4x Units (Symbolic Memory window context menu) 133
 --64bit_doubles (C-SPY command line option) 223
 --64k_flash (C-SPY command line option) 223