

IAR Embedded Workbench[®]

C-SPY[®] Debugging Guide

for Advanced RISC Machines Ltd's
ARM[®] Cores



UCSARM-3

 **IAR**
SYSTEMS

COPYRIGHT NOTICE

© 1999–2011 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, The Code to Success, IAR KickStart Kit, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB. J-Link and J-Trace are trademarks licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

ARM and Thumb are registered trademarks of Advanced RISC Machines Ltd. EmbeddedICE is a trademark of Advanced RISC Machines Ltd. OCDemon is a trademark of Macraigor Systems LLC. μ C/OS-II is a trademark of Micrium, Inc. CMX-RTX is a trademark of CMX Systems, Inc. ThreadX is a trademark of Express Logic. RTXC is a trademark of Quadros Systems. Fusion is a trademark of Unicoi Systems.

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

Third edition: October 2011

Part number: UCSARM-3

This guide applies to version 6.3x of IAR Embedded Workbench® for ARM®.

Internal reference: M11, Too6.3, IMAE.

Brief contents

Tables	17
Figures	19
Preface	25
The IAR C-SPY Debugger	31
Getting started using C-SPY	45
Executing your application	65
Working with variables and expressions	83
Using breakpoints	107
Monitoring memory and registers	139
Collecting and using trace data in the JTAGjet driver	161
Collecting and using trace data	169
Using the profiler	213
Debugging in the power domain	225
Code coverage	241
Interrupts	245
Using C-SPY macros	267
The C-SPY Command Line Utility—cspybat	321
Debugger options	353
Additional information on C-SPY drivers	385
Using flash loaders	397
Index	403

Contents

Tables	17
Figures	19
Preface	25
Who should read this guide	25
How to use this guide	25
What this guide contains	26
Other documentation	27
User and reference guides	27
The online help system	28
Web sites	28
Document conventions	28
Typographic conventions	29
Naming conventions	29
The IAR C-SPY Debugger	31
Introduction to C-SPY	31
An integrated environment	31
General C-SPY debugger features	32
RTOS awareness	33
Debugger concepts	34
C-SPY and target systems	34
The debugger	35
The target system	35
The application	35
C-SPY debugger systems	36
The ROM-monitor program	36
Third-party debuggers	36
C-SPY plugin modules	36
C-SPY drivers overview	37
Differences between the C-SPY drivers	38

The IAR C-SPY Simulator	39
Features	39
Selecting the simulator driver	39
The C-SPY hardware debugger drivers	40
Communication overview	40
Hardware installation	43
USB driver installation	43
Getting started using C-SPY	45
Setting up C-SPY	45
Setting up for debugging	45
Executing from reset	46
Using a setup macro file	47
Selecting a device description file	47
Loading plugin modules	48
Starting C-SPY	48
Starting the debugger	48
Loading executable files built outside of the IDE	48
Starting a debug session with source files missing	49
Loading multiple images	50
Adapting for target hardware	50
Modifying a device description file	51
Initializing target hardware before C-SPY starts	51
Remapping memory	52
An overview of the debugger startup	53
Debugging code in flash	53
Debugging code in RAM	54
Running example projects	54
Running an example project	55
Reference information on starting C-SPY	56
C-SPY Debugger main window	56
Images window	62
Get Alternative File dialog box	63

Executing your application	65
Introduction to application execution	65
Briefly about application execution	65
Source and disassembly mode debugging	65
Single stepping	66
Running the application	68
Highlighting	69
Call stack information	69
Terminal input and output	70
Debug logging	70
Reference information on application execution	71
Disassembly window	71
Call Stack window	75
Terminal I/O window	77
Terminal I/O Log File dialog box	78
Debug Log window	79
Log File dialog box	80
Autostep settings dialog box	81
Working with variables and expressions	83
Introduction to working with variables and expressions	83
Briefly about working with variables and expressions	83
C-SPY expressions	84
Limitations on variable information	86
Procedures for working with variables and expressions	87
Using the windows related to variables and expressions	88
Viewing assembler variables	88
Getting started using data logging	89
Getting started using event logging	90
Reference information on working with variables and expressions	91
Auto window	92
Locals window	92
Watch window	93

Live Watch window	94
Statics window	95
Quick Watch window	98
Symbols window	99
Resolve Symbol Ambiguity dialog box	100
Data Log window	101
Data Log Summary window	103
Event Log window	104
Event Log Summary window	105
Using breakpoints	107
Introduction to setting and using breakpoints	107
Reasons for using breakpoints	107
Briefly about setting breakpoints	108
Breakpoint types	108
Breakpoint icons	110
Breakpoints in the C-SPY simulator	110
Breakpoints in the C-SPY hardware drivers	111
Breakpoint consumers	111
Breakpoints options	112
Breakpoints on exception vectors	112
Setting breakpoints in __ramfunc declared functions	112
Procedures for setting breakpoints	113
Various ways to set a breakpoint	113
Toggling a simple code breakpoint	113
Setting breakpoints using the dialog box	114
Setting a data breakpoint in the Memory window	115
Setting breakpoints using system macros	116
Setting a breakpoint on an exception vector	117
Useful breakpoint hints	117
Reference information on breakpoints	118
Breakpoints window	119
Breakpoint Usage window	121
Code breakpoints dialog box	122

JTAG Watchpoints dialog box	124
Log breakpoints dialog box	127
Data breakpoints dialog box	128
Data Log breakpoints dialog box	130
Breakpoints options	132
Immediate breakpoints dialog box	134
Vector Catch dialog box	135
Enter Location dialog box	135
Resolve Source Ambiguity dialog box	137
Monitoring memory and registers	139
Introduction to monitoring memory and registers	139
Briefly about monitoring memory and registers	139
C-SPY memory zones	141
Stack display	141
Memory access checking	142
Reference information on memory and registers	143
Memory window	143
Memory Save dialog box	147
Memory Restore dialog box	148
Fill dialog box	148
Symbolic Memory window	150
Stack window	152
Register window	155
Memory Access Setup dialog box	157
Edit Memory Access dialog box	159
Collecting and using trace data in the JTAGjet driver	161
Using JTAGjet trace	161
Briefly about using JTAGjet trace	161
The JTAGjet Trace window	162
Trace view field configuration dialog box	164
Trace search query dialog box	165
ETM Control dialog box	167
ETM Configuration dialog box	168

Collecting and using trace data	169
Introduction to using trace	169
Reasons for using trace	169
Briefly about trace	170
Requirements for using trace	171
Procedures for using trace	172
Getting started with trace in the C-SPY simulator	172
Getting started with ETM trace	173
Getting started with SWO trace	173
Setting up concurrent use of ETM and SWO	174
Trace data collection using breakpoints	174
Searching in trace data	175
Browsing through trace data	176
Reference information on trace	176
ETM Trace Settings dialog box	177
SWO Trace Window Settings dialog box	179
SWO Configuration dialog box	181
Trace window	184
Trace Save dialog box	188
Function Trace window	189
Timeline window	190
Viewing Range dialog box	197
Trace Start breakpoints dialog box (simulator)	198
Trace Stop breakpoints dialog box (simulator)	199
Trace Start breakpoints dialog box	200
Trace Stop breakpoints dialog box	203
Trace Filter breakpoints dialog box	205
Trace Expressions window	208
Find in Trace dialog box	209
Find in Trace window	210
Using the profiler	213
Introduction to the profiler	213
Reasons for using the profiler	213

Briefly about the profiler	214
Requirements for using the profiler	215
Procedures for using the profiler	215
Getting started using the profiler on function level	216
Getting started using the profiler on instruction level	216
Selecting a time interval for profiling information	217
Reference information on the profiler	218
Function Profiler window	219
Debugging in the power domain	225
Introduction to power debugging	225
Reasons for using power debugging	225
Briefly about power debugging	225
Requirements for power debugging	227
Optimizing your source code for power consumption	227
Waiting for device status	227
Software delays	227
DMA versus polled I/O	228
Low-power mode diagnostics	228
CPU frequency	228
Detecting mistakenly unattended peripherals	229
Peripheral units in an event-driven system	230
Finding conflicting hardware setups	231
Analog interference	231
Procedures for power debugging	232
Displaying the application's power profile and analyzing the result	232
Detecting unexpected power usage during application execution ...	233
Reference information on power debugging	234
Power Setup window	234
Power Log window	236
Code coverage	241
Introduction to code coverage	241
Reasons for using code coverage	241

Briefly about code coverage	241
Requirements for using code coverage	241
Reference information on code coverage	242
Code Coverage window	242
Interrupts	245
Introduction to interrupts	245
Briefly about the interrupt simulation system	245
Interrupt characteristics	247
Interrupt simulation states	247
C-SPY system macros for interrupt simulation	249
Target-adapting the interrupt simulation system	249
Briefly about interrupt logging	250
Procedures for interrupts	250
Simulating a simple interrupt	251
Simulating an interrupt in a multi-task system	252
Getting started using interrupt logging using C-SPY hardware drivers	253
Reference information on interrupts	253
Interrupt Setup dialog box	254
Edit Interrupt dialog box	256
Forced Interrupt window	257
Interrupt Status window	258
Interrupt Log window	260
Interrupt Log Summary window	264
Using C-SPY macros	267
Introduction to C-SPY macros	267
Reasons for using C-SPY macros	267
Briefly about using C-SPY macros	268
Briefly about setup macro functions and files	268
Briefly about the macro language	269
Procedures for using C-SPY macros	270
Registering C-SPY macros—an overview	270
Executing C-SPY macros—an overview	270

Using the Macro Configuration dialog box	271
Registering and executing using setup macros and setup files	272
Executing macros using Quick Watch	273
Executing a macro by connecting it to a breakpoint	274
Reference information on the macro language	275
Macro functions	276
Macro variables	276
Macro strings	277
Macro statements	277
Formatted output	278
Reference information on reserved setup macro function names	280
Reference information on C-SPY system macros	281
The C-SPY Command Line Utility—cspybat	321
Using C-SPY in batch mode	321
Invocation syntax	321
Output	322
Using an automatically generated batch file	323
Summary of C-SPY command line options	323
General cspybat options	323
Options available for all C-SPY drivers	323
Options available for the simulator driver	325
Options available for the C-SPY Angel debug monitor driver	325
Options available for the C-SPY GDB Server driver	325
Options available for the C-SPY IAR ROM-monitor driver	325
Options available for the C-SPY J-Link/J-Trace driver	325
Options available for the C-SPY TI Stellaris driver	326
Options available for the C-SPY TI XDS100 driver	326
Options available for the C-SPY Macraigor driver	326
Options available for the C-SPY RDI driver and the JTAGjet driver	327
Options available for the C-SPY ST-LINK driver	327
Options available for the C-SPY third-party drivers	327

Reference information on C-SPY command line options ...	327
Debugger options	353
Setting debugger options	353
Reference information on debugger options	354
Setup	355
Download	357
Extra Options	358
Images	359
Plugins	360
Reference information on C-SPY driver options	361
Angel	361
GDB Server	363
IAR ROM-monitor	364
Setup options for J-Link/J-Trace	365
Connection options for J-Link/J-Trace	369
JTAGjet	371
RDI Configuration dialog box for JTAGjet	373
Macraigor	375
RDI	377
ST-LINK	379
Setup options for TI Stellaris	380
Setup options for TI XDS100	381
Third-Party Driver options	382
Additional information on C-SPY drivers	385
Reference information on the C-SPY simulator	385
Simulator menu	386
The C-SPY GDB Server driver	387
GDB Server menu	387
The C-SPY J-Link/J-Trace driver	387
J-Link menu	388
Live watch and use of DCC	390
Terminal I/O and use of DCC	390

The C-SPY JTAGjet driver	391
JTAGjet menu	391
The C-SPY Macraigor driver	391
Macraigor JTAG menu	392
The C-SPY RDI driver	392
RDI menu	392
The C-SPY ST-LINK driver	393
ST-LINK menu	393
The C-SPY TI Stellaris driver	394
TI Stellaris menu	395
The C-SPY TI XDS100 driver	395
TI XDS100 menu	395
Using flash loaders	397
Introduction to the flash loader	397
Briefly about the flash loader	397
Setting up the flash loader(s)	397
The flash loading mechanism	398
Reference information on the flash loader	398
Flash Loader Overview dialog box	399
Flash Loader Configuration dialog box	400
Index	403

Tables

1: Typographic conventions used in this guide	29
2: Naming conventions used in this guide	29
3: Driver differences, J-Link/J-Trace and ST-LINK	38
4: Driver differences, other drivers	39
5: Available quickstart reference information	46
6: C-SPY assembler symbols expressions	85
7: Handling name conflicts between hardware registers and assembler labels	85
8: Effects of display format setting on different types of expressions	94
9: Effects of display format setting on different types of expressions	97
10: C-SPY macros for breakpoints	116
11: JTAGjet trace save formats	163
12: JTAGjet trace searching conditions	166
13: Supported graphs in the Timeline window	191
14: C-SPY driver profiling support	215
15: Project options for enabling the profiler	216
16: Project options for enabling code coverage	242
17: Timer interrupt settings	252
18: Examples of C-SPY macro variables	276
19: C-SPY setup macros	280
20: Summary of system macros	281
21: __cancelInterrupt return values	284
22: __disableInterrupts return values	286
23: __driverType return values	286
24: __emulatorSpeed return values	287
25: __enableInterrupts return values	288
26: __evaluate return values	289
27: __hwReset return values	289
28: __hwResetRunToBp return values	290
29: __hwResetWithStrategy return values	291
30: __isBatchMode return values	292
31: __jtagResetTRST return values	297

32: __loadImage return values	298
33: __openFile return values	300
34: __readFile return values	302
35: __setCodeBreak return values	306
36: __setDataBreak return values	308
37: __setLogBreak return values	309
38: __setSimBreak return values	311
39: __setTraceStartBreak return values	312
40: __setTraceStopBreak return values	314
41: __sourcePosition return values	315
42: __unloadImage return values	318
43: cspybat parameters	321
44: Options specific to the C-SPY drivers you are using	353
45: Catching exceptions	372
46: Catching exceptions	378

Figures

1: C-SPY and target systems	35
2: C-SPY driver communication overview with a debug probe or emulator	41
3: C-SPY driver communication overview without a debug probe	42
4: Get Alternative File dialog box	49
5: Debugger startup when debugging code in flash	53
6: Debugger startup when debugging code in RAM	54
7: Example applications	55
8: Debug menu	58
9: Disassembly menu	60
10: Images window	62
11: Images window context menu	63
12: Get Alternative File dialog box	63
13: C-SPY highlighting source location	69
14: C-SPY Disassembly window	71
15: Disassembly window context menu	73
16: Call Stack window	75
17: Call Stack window context menu	76
18: Terminal I/O window	77
19: Ctrl codes menu	77
20: Input Mode dialog box	78
21: Terminal I/O Log File dialog box	78
22: Debug Log window (message window)	79
23: Debug Log window context menu	79
24: Log File dialog box	80
25: Autostep settings dialog box	81
26: Viewing assembler variables in the Watch window	89
27: Auto window	92
28: Locals window	92
29: Watch window	93
30: Watch window context menu	93
31: Live Watch window	94

32: Statics window	95
33: Statics window context menu	96
34: Quick Watch window	98
35: Symbols window	99
36: Symbols window context menu	99
37: Resolve Symbol Ambiguity dialog box	100
38: Data Log window	101
39: Data Log Summary window	103
40: Event Log window	104
41: Event Log Summary window	105
42: Breakpoint icons	110
43: Modifying breakpoints via the context menu	115
44: Breakpoints window	119
45: Breakpoints window context menu	120
46: Breakpoint Usage dialog box	121
47: Code breakpoints dialog box	122
48: JTAG Watchpoints dialog box	124
49: Log breakpoints dialog box	127
50: Data breakpoints dialog box	128
51: Data Log breakpoints dialog box	130
52: Breakpoints options	132
53: Immediate breakpoints dialog box	134
54: The Vector Catch dialog box—for ARM9/Cortex-R4 versus for Cortex-M3 ..	135
55: Enter Location dialog box	135
56: Resolve Source Ambiguity dialog box	137
57: Zones in C-SPY	141
58: Memory window	143
59: Memory window context menu	145
60: Memory Save dialog box	147
61: Memory Restore dialog box	148
62: Fill dialog box	148
63: Symbolic Memory window	150
64: Symbolic Memory window context menu	151
65: Stack window	152

66: Stack window context menu	154
67: Register window	155
68: Memory Access Setup dialog box	157
69: Edit Memory Access dialog box	159
70: The JTAGjet Trace window	162
71: Trace view field configuration dialog box	164
72: Trace search query dialog box	165
73: ETM Control dialog box	167
74: ETM Configuration dialog box	168
75: ETM Trace Settings dialog box	177
76: SWO Trace Window Settings dialog box	179
77: SWO Configuration dialog box	181
78: The Trace window in the simulator	184
79: Trace Save dialog box	188
80: Function Trace window	189
81: Timeline window	190
82: Timeline window context menu for the Call Stack Graph	194
83: Viewing Range dialog box	197
84: Trace Start breakpoints dialog box	198
85: Trace Stop breakpoints dialog box	199
86: Trace Start breakpoints dialog box (J-Link/J-Trace)	200
87: Trace Stop breakpoints dialog box (J-Link/J-Trace)	203
88: Trace Filter breakpoints dialog box	205
89: Trace Expressions window	208
90: Find in Trace dialog box	209
91: Find in Trace window	210
92: Instruction count in Disassembly window	217
93: Power Graph with a selected time interval	218
94: Function Profiler window in time-interval mode	218
95: Function Profiler window	219
96: Function Profiler window context menu	222
97: Power consumption in an event-driven system	230
98: A noise spike recorded by an oscilloscope	231
99: Power Setup window	234

100: Power Setup window context menu	235
101: Power Log window	236
102: Power Log window context menu	237
103: Code Coverage window	242
104: Code coverage window context menu	244
105: Simulated interrupt configuration	247
106: Simulation states - example 1	248
107: Simulation states - example 2	248
108: Interrupt Setup dialog box	254
109: Edit Interrupt dialog box	256
110: Forced Interrupt window	257
111: Forced Interrupt window context menu	258
112: Interrupt Status window	258
113: Interrupt Log window	260
114: Interrupt Log window context menu	263
115: Interrupt Log Summary window	264
116: Macro Configuration dialog box	271
117: Quick Watch window	274
118: Debugger setup options	355
119: C-SPY Download options	357
120: Debugger extra options	358
121: Debugger images options	359
122: Debugger plugin options	360
123: C-SPY Angel options	361
124: GDB Server options	363
125: IAR ROM-monitor options	364
126: J-Link/J-Trace Setup options	365
127: J-Link/J-Trace Connection options	369
128: JTAGjet options	371
129: The RDI Configuration dialog box	373
130: Macraigor options	375
131: RDI options	377
132: ST-LINK Setup options	379
133: TI Stellaris Setup options	380

134: C-SPY Third-Party Driver options	382
135: Simulator menu	386
136: The GDB Server menu	387
137: The J-Link menu	388
138: The JTAGjet menu	391
139: The Macraigor JTAG menu	392
140: The RDI menu	392
141: The ST-LINK menu	393
142: The TI Stellaris menu	395
143: The TI XDS100 menu	395
144: Flash Loader Overview dialog box	399
145: Flash Loader Configuration dialog box	400

Preface

Welcome to the *C-SPY® Debugging Guide for ARM*. 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 ARM core.

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 ARM core (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 27.

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 for ARM*, 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 for ARM* 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.
- *Debugging in the power domain* describes hardware solutions for measuring power consumption and how you can use them from C-SPY to find source code constructions that result in power leaks.
- *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.
- *Using flash loaders* describes the flash loader, what it is and how to use it.

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 for ARM*.
- Programming for the IAR C/C++ Compiler for ARM and linking using the IAR ILINK Linker, see the *IAR C/C++ Development Guide for ARM*.
- Programming for the IAR Assembler for ARM, see the *IAR Assembler Reference Guide for ARM*.
- Using the IAR DLIB Library, see the *DLIB Library Reference information*, available in the online help system.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for ARM, see the *IAR Embedded Workbench® Migration Guide for ARM*.
- 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*.

- IAR J-Link and IAR J-Trace, refer to the *IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores*.

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.

WEB SITES

Recommended web sites:

- The Advanced RISC Machines Ltd web site, **[www.armcom](http://www.armcom.com)**, that contains information and news about the ARM cores.
- The IAR Systems web site, **www.iar.com**, that holds application notes and other product information.
- The web site of the C standardization working group, **www.open-std.org/jtc1/sc22/wg14**.
- The web site of the C++ Standards Committee, **www.open-std.org/jtc1/sc22/wg21**.
- Finally, the Embedded C++ Technical Committee web site, **www.caravan.net/ec2plus**, that 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 `arm\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench 6.n\arm\doc`.

TYPOGRAPHIC CONVENTIONS

This guide uses the following typographic conventions:





Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> Source code examples and file paths. 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. <i>arm79</i>
[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 ARM	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for ARM	the IDE
IAR C-SPY® Debugger for ARM	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator

Table 2: Naming conventions used in this guide

Brand name	Generic term
IAR C/C++ Compiler™ for ARM	the compiler
IAR Assembler™ for ARM	the assembler
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Library™	the DLIB 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 hardware debugger drivers.

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
- Optional terminal I/O emulation.

RTOS AWARENESS

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

- AVIX-RT
- CMX-RTX
- CMX-Tiny+
- eForce μ C3/Compact
- eSysTech X realtime kernel
- Express Logic ThreadX
- FreeRTOS, OpenRTOS, and SafeRTOS
- Freescale MQX
- Micrium μ C/OS-II
- Micro Digital SMX
- MISPO NORTi
- OSEK (ORTI)

- RTXC Quadros
- Segger embOS
- unicon Fusion.

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. For links to the RTOS documentation, see the release notes that are available from the **Help** menu.

Debugger concepts

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

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:

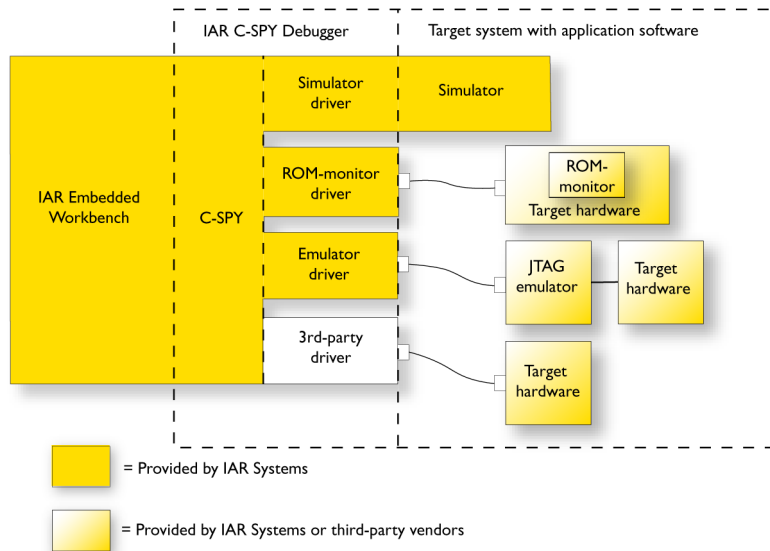


Figure 1: C-SPY and target systems

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 37.

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 ELF/DWARF, Intel-extended, or Motorola. 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.

- Peripheral simulation modules make C-SPY simulate peripheral units. Such plugin modules are not provided by IAR Systems, but can be developed and distributed by third-party suppliers.
- 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 is available with drivers for these target systems and evaluation boards:

- Simulator
- J-Link / J-Trace JTAG/SWD probes
- JTAGjet probes
- RDI (Remote Debug Interface)
- Macraigor JTAG probes
- GDB Server
- ST-LINK JTAG/SWD probe (for ST Cortex-M devices only)
- TI Stellaris JTAG/SWD interface using FTDI or ICDI (for Stellaris Cortex devices only)
- TI XDS100 JTAG interface
- P&E Microcomputer Systems. For information about this driver, see the document *Configuring IAR Embedded Workbench for ARM to use a P&E Microcomputer Systems Interface*, available in the `arm\doc` directory.
- Angel debug monitor
- IAR ROM-monitor for Analog Devices ADuC7xxx boards, and IAR Kickstart Card for Philips LPC210x.

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 382.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

This table summarizes the key differences between the Simulator, J-Link/J-Trace, and ST-LINK:

Feature	Simulator	J-Link/J-Trace	ST-LINK	Comment, for J-Link and ST-LINK
Code breakpoints	x	x	x	--
Data breakpoints	x	x	x	--
Interrupt logging	x	x	x	Cortex with SWD/SWO
Data logging	--	x	x	Cortex with SWD/SWO
Call stack trace	--	x	--	Requires ETM/ETB trace.
Power logging	--	x	--	Requires a Cortex-M device with SWO and a J-Link or J-Link Ultra debug probe.
Event logging	--	x	x	Cortex with SWD/SWO, using ITM channel
Live watch	--	x	x	Supported by Cortex devices. For ARM7/9 devices, Live watch is supported if you add a DCC handler to your application. See <i>Live watch and use of DCC</i> , page 390.
Cycle counter	x	x	x	For Cortex-M devices only.
Code coverage	x	x	x	Supported by J-Trace and J-link with ETB. For Cortex-M devices, J-Link and ST-LINK with SWO support partial code coverage. For more information about code coverage, see <i>Code coverage</i> , page 241.
Data coverage	x	--	--	--
Function/instruction profiler	x	x	x	Requires either SWD/SWO interface or ETM/ETB trace.
Trace	x	x	--	Requires ETM/ETB trace.

Table 3: Driver differences, J-Link/J-Trace and ST-LINK

This table summarizes the key differences between the Simulator and other supported hardware debugger drivers:

Feature	Simulator	JTAGjet	RDI	Mac-raigor	GDB Server	TI Stellaris	TI XDS100	Angel
Code breakpoints	x	x	x	x	x	x	x	x
Data breakpoints	x	x	x	x	x	x	--	--
Interrupt logging	x	--	--	--	--	--	--	--
Cycle counter	x	--	--	--	--	--	--	--
Code coverage	x	--	--	--	--	--	--	--
Data coverage	x	--	--	--	--	--	--	--
Function/instruction profiler	x	--	--	--	--	--	--	--
Trace	x	x	x	--	--	--	--	--

Table 4: Driver differences, other drivers

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 hardware debugger drivers

C-SPY can connect to a hardware debugger using a C-SPY hardware debugger driver as an interface.

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

COMMUNICATION OVERVIEW

There are two main communication setups, depending on the type of target system. Many of the ARM cores have built-in, on-chip debug support. Because the hardware debugger logic is built into the core, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work, other than the debug probe. For some devices that do not have such built-in, on-chip debug support, there are instead a ROM-monitor debugger solution that can be used.

Overview of a target system with a debug probe or emulator

Most target systems have an emulator, a debug probe or a debug adapter connected between the host computer and the evaluation board:

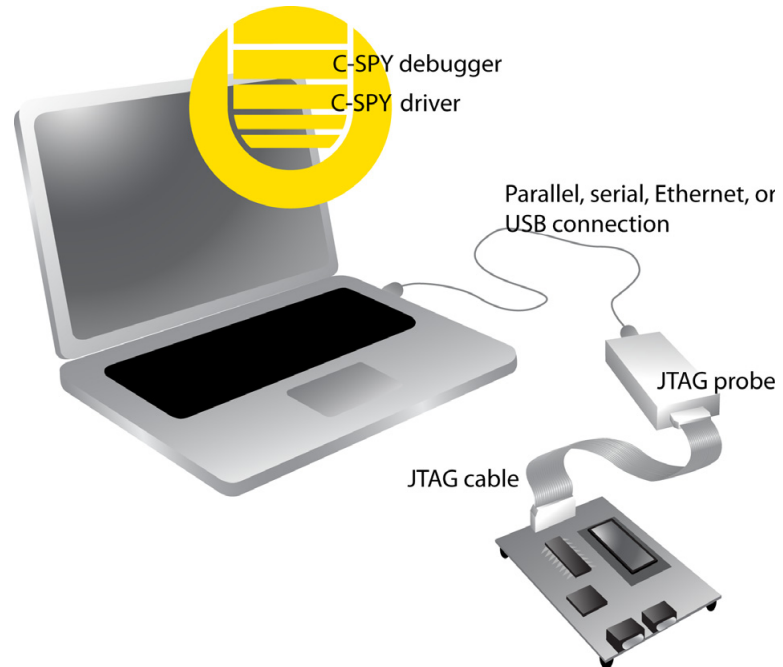


Figure 2: C-SPY driver communication overview with a debug probe or emulator

When USB connection is used, a specific USB driver must be installed before you can use the probe over the USB port. You can find the driver on the IAR Embedded Workbench for ARM installation media.

Overview of a target systems using a ROM-monitor

IAR Embedded Workbench comes with two ready-made ROM-monitors:

- Using the IAR Angel debug monitor driver, you can communicate with any device compliant with the Angel debug monitor protocol. In most cases these are evaluation boards.
- Using the IAR ROM-monitor driver, C-SPY can connect to the Analog Devices ADuC7xxx boards and the IAR Kickstart Card for Philips LPC210x. Most ROM-monitors require that the code that you want to debug is located in RAM,

because the only way you can set breakpoints and step in your application code is to download it to RAM. For some ROM-monitors, for example for Analog Devices ADuC7xxx, the code that you want to debug can be located in flash memory. To maintain debug functionality, the ROM-monitor might simulate some instructions, for example when single stepping.

The boards contain firmware (the ROM-monitor itself) that runs in parallel with your application software. The firmware receives commands from the IAR C-SPY debugger over a serial port, and controls the execution of your application.

Using the C-SPY ROM-monitor driver, C-SPY can connect to a target system equipped with a ROM-monitor located in flash memory.

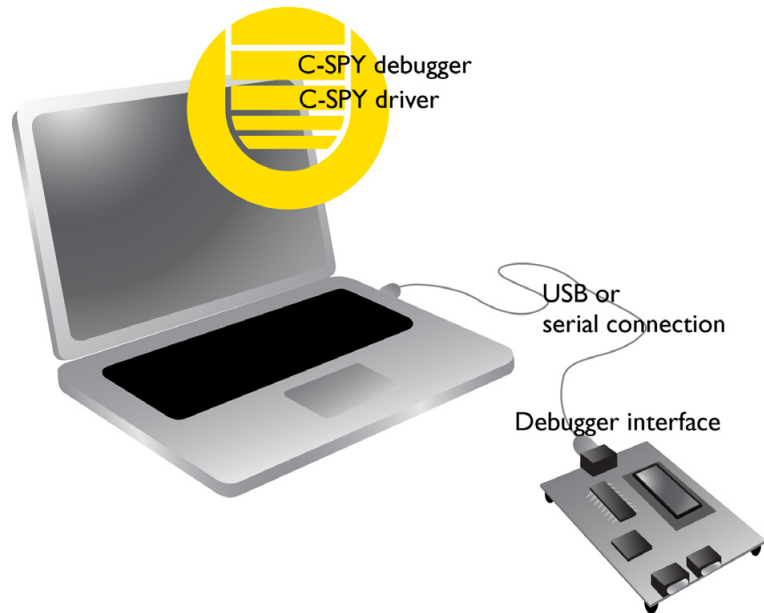


Figure 3: C-SPY driver communication overview without a debug probe

This is an inexpensive solution to debug a target, because only a serial cable is needed. All the parts of your code that you want to debug must be located in RAM. The only way you can set breakpoints and step in your application code is to download it into RAM.

For further information, see:

- The `angel_quickstart.html` file, available in the `arm\doc\infocenter` directory, or refer to the manufacturer's documentation.

- The `iar_rom_quickstart.html` file, available in the `arm\doc\infocenter` directory, or refer to the manufacturer's documentation.

HARDWARE INSTALLATION

For information about the hardware installation, see the documentation supplied with the target system from the manufacturer. The following power-up sequence is recommended to ensure proper communication between the target board, the emulator or debug probe, and C-SPY:

- 1 Connect the USB cable to the debug probe.
- 2 Connect the probe to the target board.
- 3 Power up the debug probe, if it is not powered via USB.
- 4 Power up the target board.
- 5 Start the C-SPY debugging session.

USB DRIVER INSTALLATION

A USB driver is also needed. In some cases this driver is automatically installed, but for some probes you need to manually install it.

Installing the J-Link USB driver

Before you can use the J-Link JTAG probe over the USB port, the Segger J-Link USB driver must be installed.

- 1 Install IAR Embedded Workbench for ARM.
- 2 Use the USB cable to connect the computer and J-Link. Do not connect J-Link to the target board yet. The green LED on the front panel of J-Link will blink for a few seconds while Windows searches for a USB driver.

Run the `InstDrivers.exe` application, which is located in the product installation in the `arm\drivers\JLink` directory.

Once the initial setup is completed, you will not have to install the driver again.

Note that J-Link will continuously blink until the USB driver has established contact with the J-Link probe. When contact has been established, J-Link will start with a steady light to indicate that it is connected.

Installing the ST-LINK USB driver for ST-LINK ver. 2

Before you can use the ST-LINK version 2 JTAG probe over the USB port, the ST-LINK USB driver must be installed.

- 1 Install IAR Embedded Workbench for ARM.
- 2 Use the USB cable to connect the computer and ST-LINK. Do not connect ST-LINK to the target board yet.

Because this is the first time ST-LINK and the computer are connected, Windows will open a dialog box and ask you to locate the USB driver. The USB driver can be found in the product installation in the `arm\drivers\ST-Link` directory:
`ST-Link_V2_USBdriver.exe`.

Once the initial setup is completed, you will not have to install the driver again.

Installing the TI Stellaris USB driver

Before you can use the TI Stellaris JTAG interface using FTDI or ICDI over the USB port, the Stellaris USB driver must be installed.

- 1 Install IAR Embedded Workbench for ARM.
- 2 Use the USB cable to connect the computer to the TI board.

Because this is the first time the Stellaris JTAG interface and the computer are connected, Windows will open a dialog box and ask you to locate the USB driver. There are different USB drivers for FTDI and ICDI. The drivers can be found in the product installation in the `arm\drivers\StellarisFTDI` and the `arm\drivers\StellarisICDI` directories, respectively.

Once the initial setup is completed, you will not have to install the driver again.

Installing the TI XDS100 USB driver

Before you can use the TI XDS100 JTAG interface over the USB port, the TI XDS100 USB driver must be installed.

- 1 Install IAR Embedded Workbench for ARM.
- 2 Install the TI XDS100 package which can be found in the `arm\drivers\ti-xds` directory. It is recommended to choose the suggested installation directory.
- 3 Use the USB cable to connect the computer to the TI board.

Configuring the OpenOCD Server

For further information, see the `gdbserve_quickstart.html` file, available in the `arm\doc\infocenter` directory, or refer to the manufacturer's documentation.

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
- An overview of the debugger startup
- 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

- I Install a USB driver if your C-SPY driver requires it. For more information, see:

- *Installing the J-Link USB driver*, page 43
- *Installing the ST-LINK USB driver for ST-LINK ver. 2*, page 44
- *Installing the TI Stellaris USB driver*, page 44
- *Installing the TI XDS100 USB driver*, page 44
- *Configuring the OpenOCD Server*, page 44.

- 2 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.
- 3 In the **Category** list, select the appropriate C-SPY driver and make your settings.
For information about these options, see *Debugger options*, page 353.
- 4 Click **OK**.
- 5 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 for ARM*.

The following documents containing information about how to set up various debugging systems are available in the `arm\doc\infocenter` subdirectory:

File	Debugger system
<code>rdi_quickstart.html</code>	Quickstart reference for RDI-controlled JTAG debug interfaces
<code>gdbserver_quickstart.html</code>	Quickstart reference for a GDB Server using OpenOCD together with STR9-comStick
<code>angel_quickstart.html</code>	Quickstart reference for Angel ROM-monitors and JTAG interfaces
<code>iar_rom_quickstart.html</code>	Quickstart reference for IAR ROM-monitor

Table 5: Available quickstart reference information

See also *Adapting for target hardware*, page 50.

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. The reset address is set by C-SPY.

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 268. For an example of how to use a setup macro file, see the chapter *Initializing target hardware before C-SPY starts*, page 51.

To register a setup macro file:

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

SELECTING A DEVICE DESCRIPTION FILE

C-SPY uses device description files to handle device-specific information. Device description files can be of two different formats—IAR Systems device description files or CMSIS System View Description files (SVD).

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. IAR Systems device description files are provided in the `arm\config` directory and they have the filename extension `ddf`.

For more information about device description files, see *Adapting for target hardware*, page 50.

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 360.

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:

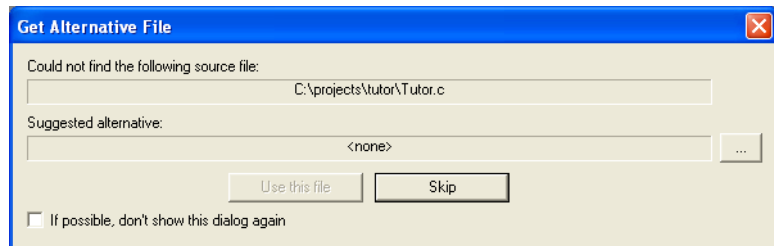


Figure 4: Get Alternative File dialog box

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 63.

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 359.
- 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 270.

To display a list of loaded images:

Choose **Images** from the **View** menu. The Images window is displayed, see *Images window*, page 62.

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
- Remapping memory.

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 47. They contain device-specific information such as:

- Definitions of registers in peripheral units and groups of these
- Interrupt definitions (for Cortex-M devices only); see *Interrupts*, page 245.

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

The syntax of the device description files is described in the *IAR Embedded Workbench for ARM device description file format* guide (EWARM_DDFFormat.pdf) located in the arm\doc directory.

For information about how to load a device description file, see *Selecting a device description file*, page 47.

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.

REMAPPING MEMORY

A common feature of many ARM-based processors is the ability to remap memory. After a reset, the memory controller typically maps address zero to non-volatile memory, such as flash. By configuring the memory controller, the system memory can be remapped to place RAM at zero and non-volatile memory higher up in the address map. By doing this, the exception table will reside in RAM and can be easily modified when you download code to the target hardware.

You must configure the memory controller before you download your application code. You can do this best by using a C-SPY macro function that is executed before the code download takes place—`execUserPreload()`. The macro function `__writeMemory32()` will perform the necessary initialization of the memory controller.

The following example illustrates a macro used for remapping memory on the Atmel AT91SAM7S256 chip, similar mechanisms exist in processors from other ARM vendors.

```
execUserPreload()
{
    // REMAP command
    // Writing 1 to MC_RCR (MC Remap Control Register)
    // will toggle remap bit.
    __writeMemory32(0x00000001, 0xFFFFFFFF00, "Memory");
}
```

Note that the setup macro `execUserReset()` might have to be defined in the same way to reinitialize the memory mapping after a C-SPY reset. This can be needed if you have set up your hardware debugger system to do a hardware reset on C-SPY reset, for example by adding `__hwReset()` to the `execUserReset()` macro.

For instructions on how to install a macro file in C-SPY, see *Registering and executing using setup macros and setup files*, page 272. For information about the macro functions used, see *Reference information on C-SPY system macros*, page 281.

An overview of the debugger startup

To make it easier to understand and follow the startup flow, the following figures show the flow of actions performed by C-SPY, and by the target hardware, as well as the execution of any predefined C-SPY setup macros. There is one figure for debugging code located in flash and one for debugging code located in RAM.

For more information about C-SPY system macros, see the chapter *Using C-SPY macros* available in this guide.

DEBUGGING CODE IN FLASH

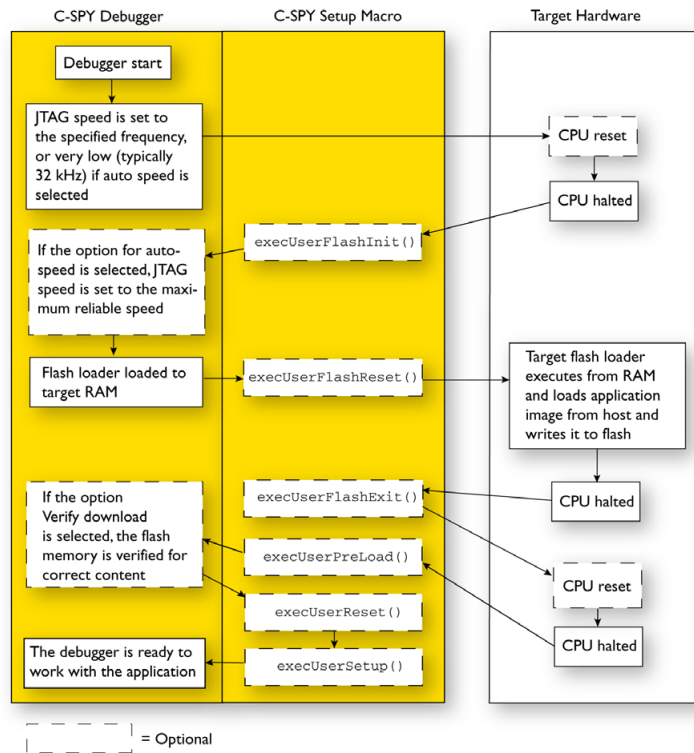


Figure 5: Debugger startup when debugging code in flash

DEBUGGING CODE IN RAM

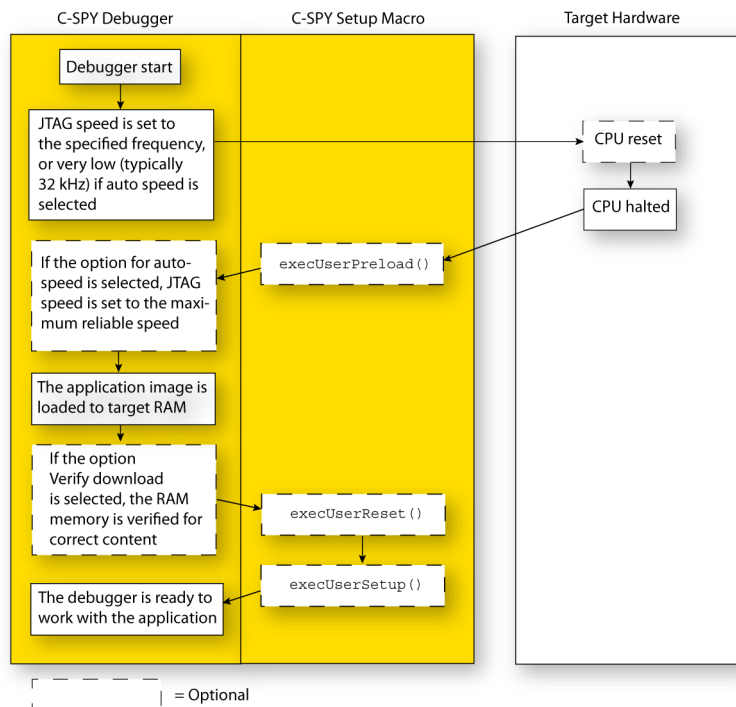


Figure 6: Debugger startup when debugging code in RAM

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 `arm\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.

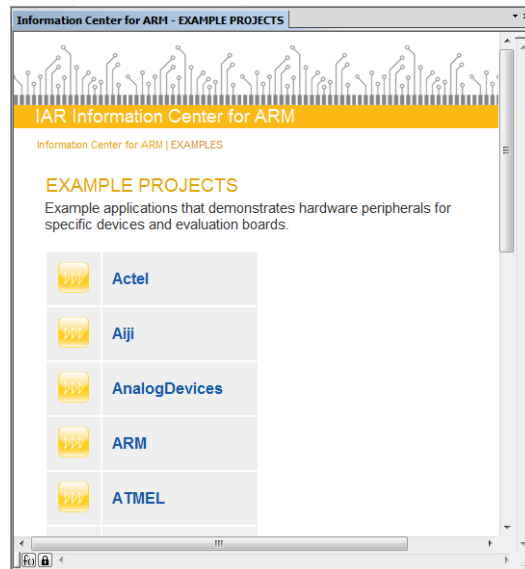


Figure 7: Example applications

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 **General Options>Target>Processor variant** 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 353.

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.



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 56
- *Images window*, page 62
- *Get Alternative File dialog box*, page 63

See also:

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

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 58. Most of the commands are also available as icon buttons on the debug toolbar.
Disassembly	Provides commands for controlling the disassembly processor mode; see <i>Disassembly window</i> , page 71.
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 386.
GDB Server	Provides commands specific to the C-SPY GDB Server driver. This menu is only available when the driver is used; see <i>GDB Server menu</i> , page 387.
J-Link	Provides commands specific to the C-SPY J-Link driver. This menu is only available when the driver is used; see <i>J-Link menu</i> , page 388.
TI Stellaris	Provides commands specific to the C-SPY TI Stellaris driver. This menu is only available when the driver is used; see <i>TI Stellaris menu</i> , page 395.
TI XDS100	Provides commands specific to the TI XDS100 driver. This menu is only available when the driver is used; see <i>TI XDS100 menu</i> , page 395.
JTAG	Provides commands specific to the C-SPY Macraigor driver. This menu is only available when the driver is used; see <i>Macraigor JTAG menu</i> , page 392.
JTAGjet	Provides commands specific to the JTAGjet driver. This menu is only available when the driver is used; see <i>JTAGjet menu</i> , page 391.
RDI	Provides commands specific to the C-SPY RDI driver. This menu is only available when the driver is used; see <i>RDI menu</i> , page 392.

ST-LINK

Provides commands specific to the C-SPY ST-LINK driver. This menu is only available when the driver is used; see *ST-LINK menu*, page 393.

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.

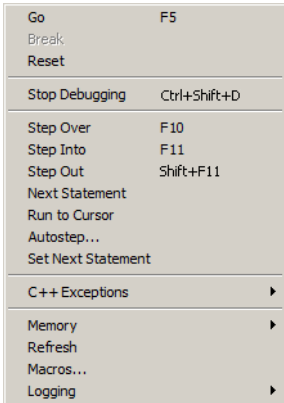




Figure 8: Debug menu


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 autostepping, see *Autostep settings dialog box*, page 81.

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**

Specifies that the execution shall break when the target application executes a `throw` statement.

To use this feature, your application must be built with the option **Library low-level interface implementation** selected and the language option C++ for Standard C++.

**C++ Exceptions>
Break on Uncaught
Exception**

Specifies that the execution shall break when the target application throws an exception that is not caught by any matching `catch` statement.

To use this feature, your application must be built with the option **Library low-level interface implementation** selected and the language option C++ for Standard C++.

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 147.

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 148.

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 <i>Using the Macro Configuration dialog box</i> , page 271.
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 <i>Log File dialog box</i> , page 80.
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 <i>Terminal I/O Log File dialog box</i> , page 78.

Disassembly menu

The **Disassembly** menu is available when C-SPY is running. This menu provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

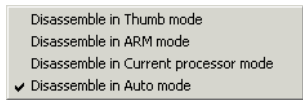


Figure 9: Disassembly menu

Use the commands on the menu to select which disassembly mode to use.

Note: After changing disassembly mode, use the **Refresh** command on the **Debug** menu to refresh the view of the Disassembly window contents.

These commands are available:

Disassemble in Thumb mode	Disassembles your application in Thumb mode.
Disassemble in ARM mode	Disassembles your application in ARM mode.
Disassemble in Current processor mode	Disassembles your application in the current processor mode.
Disassemble in Auto mode	Disassembles your application in automatic mode. This is the default option.

See also *Disassembly window*, page 71.

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
- Live Watch 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, Live Watch, 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.

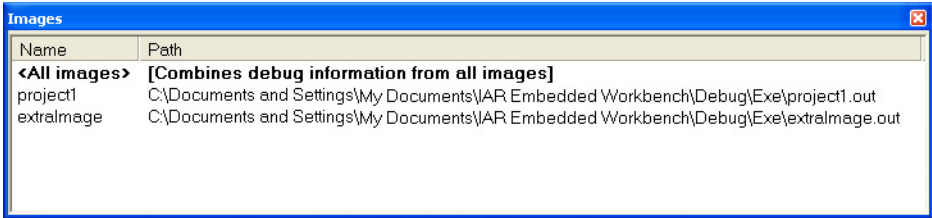


Figure 10: Images window

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 11: 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 50
- *Images*, page 359
- *__loadImage*, page 297.

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.

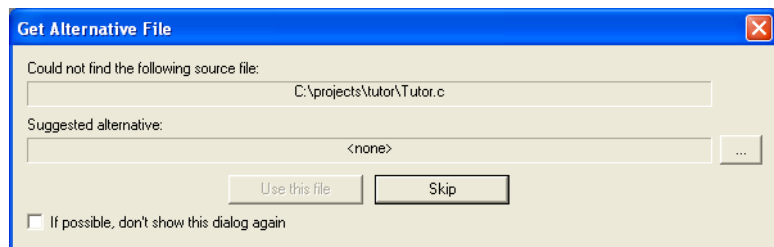


Figure 12: Get Alternative File dialog box

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 49.

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 81.

If your application contains an exception that is caught outside the code which would normally be executed as part of a step, C-SPY terminates the step at the *catch* statement.

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.

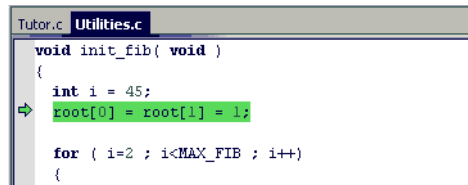


Figure 13: C-SPY highlighting source location

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 *IAR Assembler Reference Guide for ARM*.

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 77 and *Terminal I/O Log File dialog box*, page 78.

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 71
- *Call Stack window*, page 75
- *Terminal I/O window*, page 77
- *Terminal I/O Log File dialog box*, page 78
- *Debug Log window*, page 79
- *Log File dialog box*, page 80
- *Autostep settings dialog box*, page 81.

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

Disassembly window

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

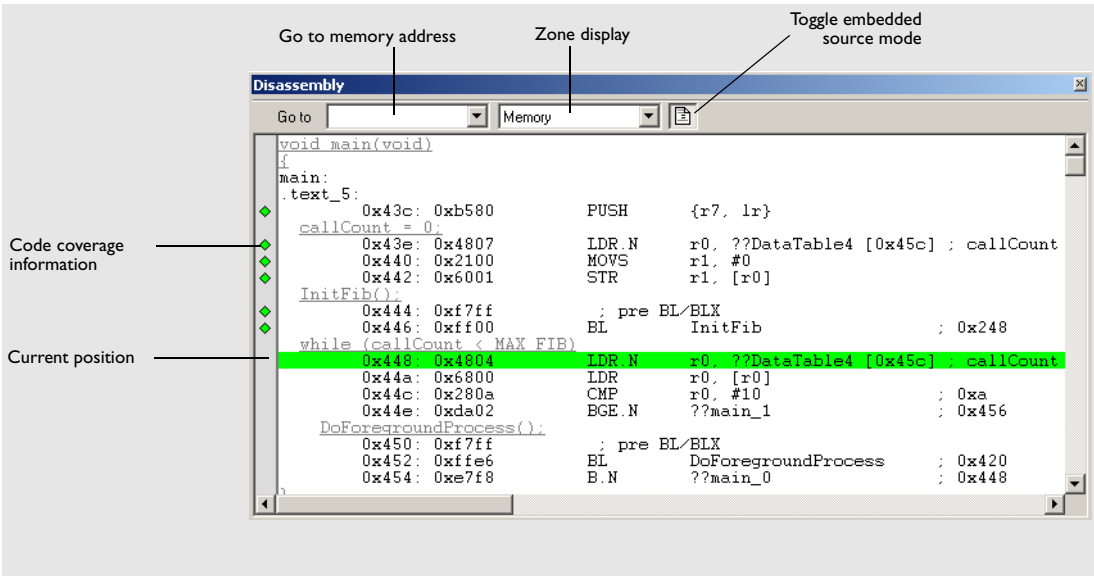


Figure 14: C-SPY Disassembly window

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 141.
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 107.
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:

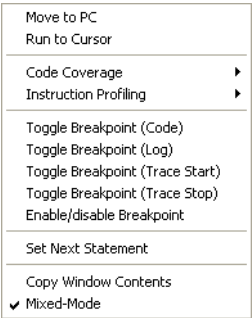


Figure 15: Disassembly window context menu

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

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

Enable, toggles code coverage on or off.

Show, toggles the display of code coverage on or off.

Executed code is indicated by a green diamond.

Clear, clears all code coverage information.
- 73

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 122.
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 127.
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 (simulator)</i> , page 198.
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 (simulator)</i> , page 199.
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.

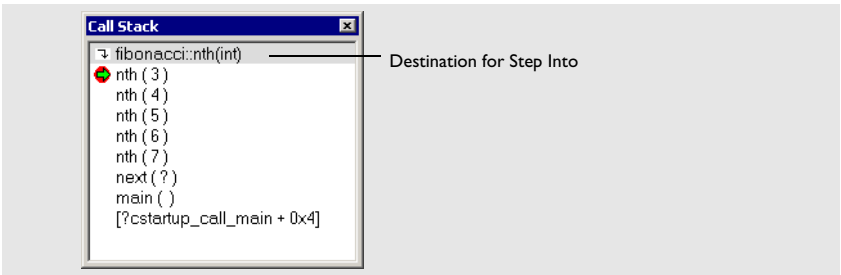


Figure 16: Call Stack window

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:

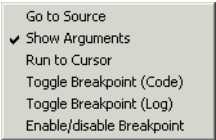


Figure 17: Call Stack window context menu

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.

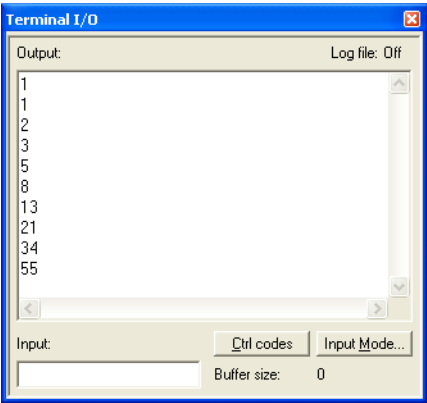


Figure 18: Terminal I/O window

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

To use this window, you must:

- 1 Build your application with the option **Semihosted** or the **IAR breakpoint** option.

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.

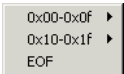


Figure 19: Ctrl codes menu

Input Mode

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

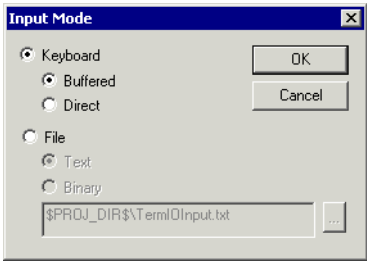


Figure 20: Input Mode dialog box

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

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**.

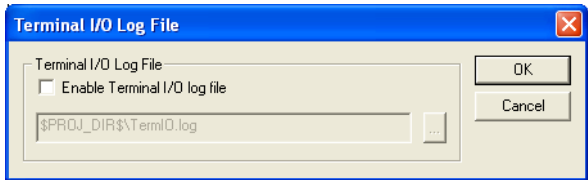


Figure 21: Terminal I/O Log File dialog box

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**.

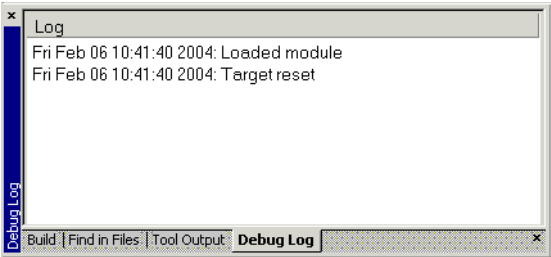


Figure 22: Debug Log window (message window)

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 for ARM*.

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:

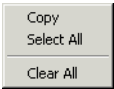


Figure 23: Debug Log window context menu

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**.

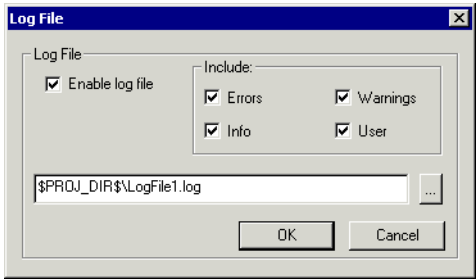


Figure 24: Log File dialog box

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`).

Autostep settings dialog box

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

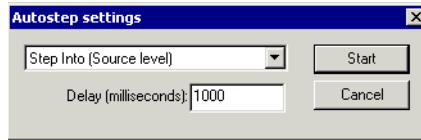


Figure 25: Autostep settings dialog box

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
- Procedures for 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.

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 Live Watch window** repeatedly samples and displays the values of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.

- 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 Data Log window and the Data Log Summary window display logs of accesses up to four different memory locations or areas you choose by setting Data Log breakpoints. Data logging can help you locate frequently accessed data. You can then consider whether you should place that data in more efficient memory. Data logs are supported by the C-SPY J-Link/J-Trace driver and the C-SPY ST-LINK driver.
 - The Event Log window and the Event Log Summary window display *event logs* produced when the execution passes specific positions in your application code. The Timeline window graphically displays these event logs correlated to a common time-axis. Event logging can help you to analyze program flow and inspect data correlated to a certain position in your application code. Event logs are supported by the C-SPY J-Link/J-Trace driver and the C-SPY ST-LINK driver.
- The Cortex ITM data channels are used for passing events from a running application to the C-SPY Event log system. There are predefined preprocessor macros that you can use in your application source code. An Event log will be generated every time such macros are passed during program execution. You can pass a value with each event. Typically, this value can be either an identifier or the content of a variable or a register (for example, the stack pointer). The value can be written in 8, 16, or 32-bit format. Using a smaller size will reduce the bandwidth needed on the SWO pin. Events can be generated with or without an associated PC (program counter) value, the PC value makes it possible for the debugger to correlate the event to the executed code.
- 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 169.

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)
```

In case you have a static variable with the same name declared in several different functions

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 R4–R15, 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 51.

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 6: 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 7: 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 155.

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 269.

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 275.

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**.

Procedures for working with variables and expressions

This section gives you step-by-step descriptions about how to work with variables and expressions.

More specifically, you will get information about:

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

USING THE WINDOWS RELATED TO VARIABLES AND EXPRESSIONS

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

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



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

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

VIEWING ASSEMBLER VARIABLES

An assembler label does not convey any type information at all, which means C-SPY cannot easily display data located at that label without getting extra information. To view data conveniently, C-SPY by default treats all data located at assembler labels as variables of type `int`. However, in the Watch, Quick Watch, and Live 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:

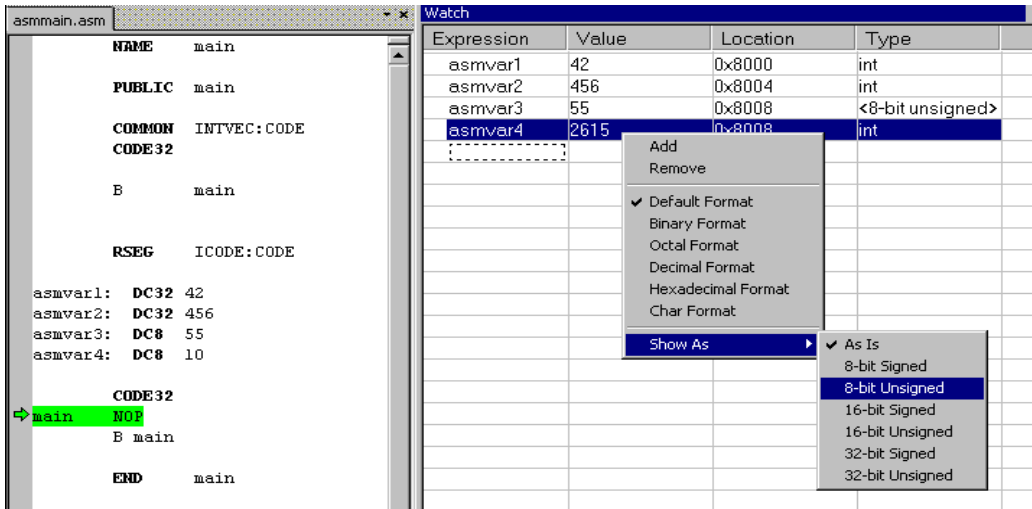


Figure 26: Viewing assembler variables in the Watch window

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.

GETTING STARTED USING DATA LOGGING

- 1 To set up for data logging, choose **J-Link>SWO Configuration** or **ST-LINK>SWO Configuration**, respectively. In the dialog box, set up the serial-wire output communication channel for trace data. Note specifically the **CPU clock** option. The CPU clock can also be set up on the **Project>Options>ST-LINK** page.
- 2 In the Breakpoints or Memory window, right-click and choose **New Breakpoints>Data Log** to open the breakpoints dialog box. Set a Data Log breakpoint on the data you want to collect log information for.
- 3 Choose **C-SPY driver>Data Log** to open the Data Log window. Optionally, you can also choose:
 - **C-SPY driver>Data Log Summary** to open the Data Log Summary window
 - **C-SPY driver>Timeline** to open the Timeline window to view the Data Log graph.

- 4 From the context menu, available in the Data Log window, choose **Enable** to enable the logging.
- 5 In the **SWO Configuration** dialog box, you can notice in the Data Log Evens area that Data Logs are enabled. Choose which level of logging you want:
 - PC only
 - PC + data value + base addr
 - Data value + exact addr
- 6 Start executing your application program to collect the log information.
- 7 To view the data log information, look in any of the Data Log, Data Log Summary, or the Data graph in the Timeline window.
- 8 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
- 9 To disable data and interrupt logging, choose **Disable** from the context menu in each window where you have enabled it.

GETTING STARTED USING EVENT LOGGING

- 1 To specify the position in your application source code that you want to generate event logs for, use the predefined preprocessor macros in `arm_itm.h` (located in `arm\inc\c`). In your application source code, write (for example):

```
#include <arm_itm.h>
void func(void)
{
    ITM_EVENT8_WITH_PC(1, 25);
    ITM_EVENT32_WITH_PC(2, __get_PSP());
}
```

The first line sends an event with the value 25 to channel 1. The second line sends an event with the current value of the stack pointer to channel 2, which means that C-SPY can display the stack pointer at a code position of your choice. When these source lines are passed during program execution, events will be generated and visualized by C-SPY, which means that you can further analyze them.

- 2 To view event log information, you can choose between these alternatives:
 - Choose *C-SPY driver*>**Timeline** to open the Timeline window and choose **Enable** from the context menu. You can now view events for each channel as a graph (Event Log Graph).
 - Choose *C-SPY driver*>**Event Log** to open the Event Log window and choose **Enable** from the context menu. You can now view the events for each channel as numbers.

- Choose *C-SPY driver* > **Event Log Summary** to open the Event Log Summary window and choose **Enable** from the context menu. You will now get a summary of all event logs.

Note: Whenever the Event Log Graph or the Event Log window is enabled, you can at any time enable also the Event Log Summary window to get a summary. However, if you have enabled the Event Log Summary window, but not the Event Log window or the Event Log Graph in the Timeline window, you can get a summary but not detailed information about event logs.

- 3 To change the display format (you can choose between displaying values in hexadecimal or in decimal format), select the event graph for which you want to change the format in the Timeline window. Right-click and choose the display format of your choice from the context menu. Note that this setting affects also the Event Log window and the Event Log Summary window.
- 4 Start executing your application program to collect the log information.
- 5 To view the event log information, look at either the Event Log, the Event Log Summary, or the event graph for the specific channel in the Timeline window.
- 6 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window.
- 7 To disable event logging, choose **Disable** from the context menu in each window where you have enabled it.

Reference information on working with variables and expressions

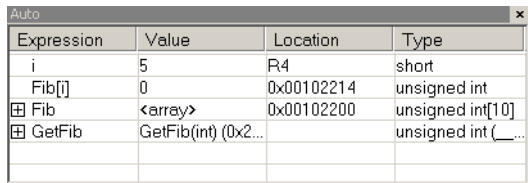
This section gives reference information about these windows and dialog boxes:

- *Auto window*, page 92
- *Locals window*, page 92
- *Watch window*, page 93
- *Live Watch window*, page 94
- *Statics window*, page 95
- *Quick Watch window*, page 98
- *Symbols window*, page 99
- *Resolve Symbol Ambiguity dialog box*, page 100
- *Data Log window*, page 101
- *Data Log Summary window*, page 103
- *Event Log window*, page 104
- *Event Log Summary window*, page 105.

For trace-related reference information, see *Reference information on trace*, page 176.

Auto window

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



Expression	Value	Location	Type
i	5	R4	short
Fib[i]	0	0x00102214	unsigned int
Fib	<array>	0x00102200	unsigned int[10]
GetFib	GetFib(int) (0x2...		unsigned int (...

Figure 27: Auto window

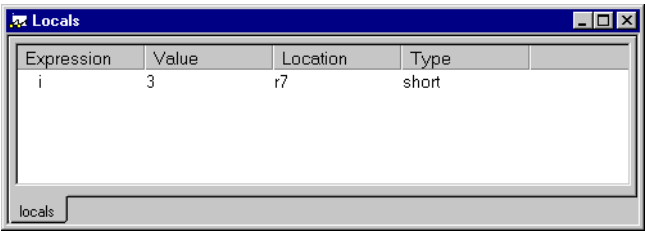
This window displays a useful selection of variables and expressions in, or near, the current statement. Every time execution in C-SPY stops, the values in the Auto window are recalculated. Values that have changed since the last stop are highlighted in red.

Context menu

For more information about the context menu, see *Watch window*, page 93.

Locals window

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



Expression	Value	Location	Type
i	3	r7	short

Figure 28: 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 93.

Watch window

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

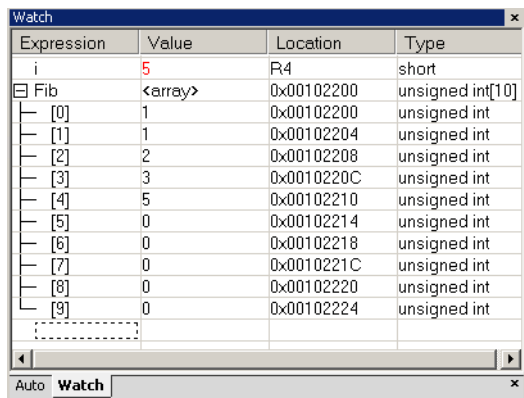


Figure 29: Watch window

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

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

Context menu

This context menu is available:

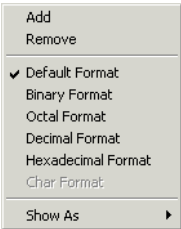


Figure 30: Watch window context menu

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 8, <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 Watch window contains a variable.
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 88.

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 8: Effects of display format setting on different types of expressions

Live Watch window

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

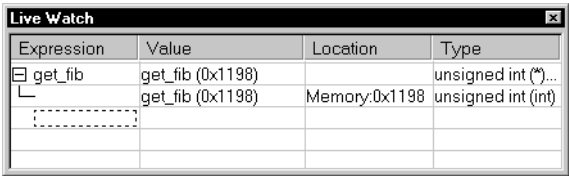


Figure 31: Live Watch window

This window repeatedly samples and displays the value of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.

This window can only be used for hardware target systems supporting this feature.

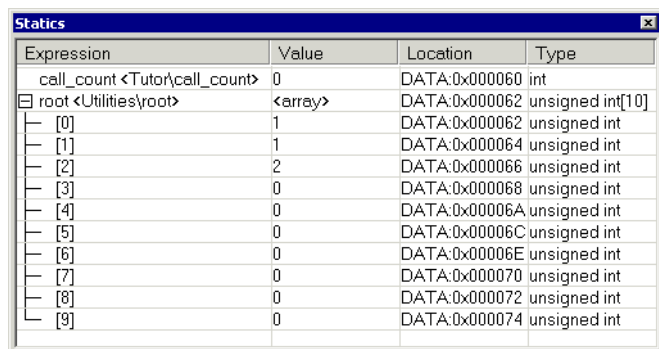
Context menu

For more information about the context menu, see *Watch window*, page 93.

In addition, the menu contains the **Options** command, which opens the **Debugger** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution.

Statics window

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



Expression	Value	Location	Type
call_count <Tutorial\call_count>	0	DATA:0x000060	int
root <Utilities\root>	<array>	DATA:0x000062	unsigned int[10]
[0]	1	DATA:0x000062	unsigned int
[1]	1	DATA:0x000064	unsigned int
[2]	2	DATA:0x000066	unsigned int
[3]	0	DATA:0x000068	unsigned int
[4]	0	DATA:0x00006A	unsigned int
[5]	0	DATA:0x00006C	unsigned int
[6]	0	DATA:0x00006E	unsigned int
[7]	0	DATA:0x000070	unsigned int
[8]	0	DATA:0x000072	unsigned int
[9]	0	DATA:0x000074	unsigned int

Figure 32: 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:

- I 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> <p>This column is editable.</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:

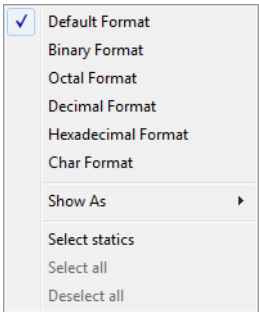


Figure 33: Statics window context menu

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 8, <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 9: 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.

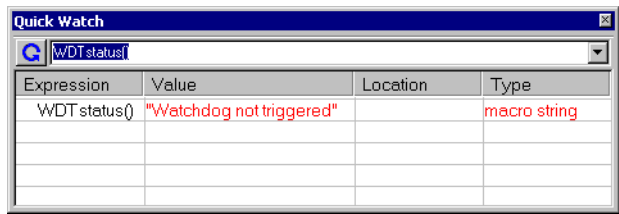


Figure 34: Quick Watch window

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 273.

Context menu

For more information about the context menu, see *Watch window*, page 93.

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.

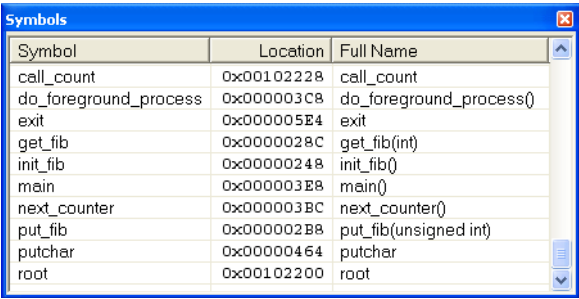


Figure 35: Symbols window

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:

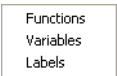


Figure 36: Symbols window context menu

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.

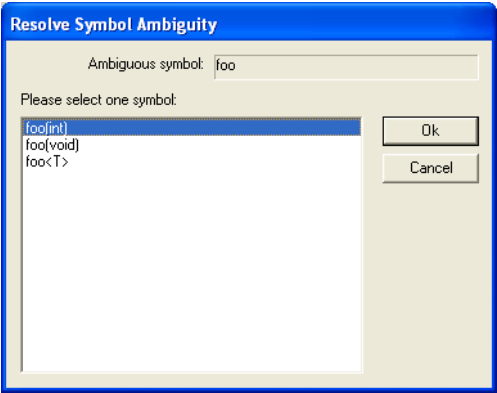


Figure 37: Resolve Symbol Ambiguity dialog box

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.

Data Log window

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

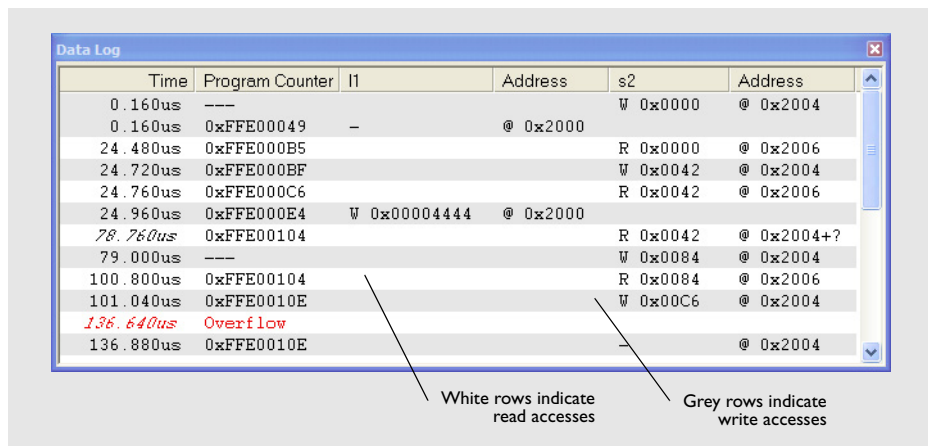


Figure 38: Data Log window

To use the Data Log window, you need:

- A J-Link debug probe, an ST-LINK debug probe, or a J-Trace debug probe. For J-Trace, the Data Log window is available when ETM trace is disabled. The Data Log window does not display any data when ETM is enabled.
- An SWD interface between the debug probe and the target system.

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

See also *Getting started using data logging*, page 89.

Display area

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

Time

The time for the data access, based on the clock frequency specified in the **SWO Configuration** dialog box.

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

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

Cycles	<p>The number of cycles from the start of the execution until the event. This information is cleared at reset.</p> <p>If a cycle is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it.</p> <p>This column is available when you have selected Show cycles from the context menu.</p>
Program Counter*	<p>The content of the PC, that is, the address of the instruction that performed the memory access.</p> <p>If the column displays ---, the target system failed to provide the debugger with any information. If the column displays Overflow in red, the communication channel transmit to handle all data from the target system.</p>
Value	<p>Displays the access type and the value (using the access size) for the location or area you want to log accesses to. For example, if zero is read using a byte access it will be displayed as 0x00, and for a long access it will be displayed as 0x00000000.</p> <p>To specify what data you want to log accesses to, use the Data Log breakpoint dialog box. See <i>Data Log breakpoints dialog box</i>, page 130.</p>
Address	<p>The actual memory address that is accessed. For example, if only a byte of a word is accessed, only the address of the byte is displayed. The address is calculated as base address + offset, where the base address is retrieved from the Data Log breakpoint dialog box and the offset is retrieved from the logs. If the log from the target system does not provide the debugger with an offset, the offset contains + ?. If you want the offset to be displayed, select the Value + exact addr option in the SWO Setup dialog box.</p>

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

Context menu

See *Interrupt Log window context menu*, page 263.

Data Log Summary window

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

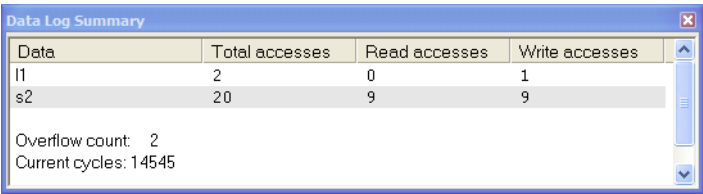


Figure 39: Data Log Summary window

To use the Data Log Summary window, you need:

- A J-Link debug probe, an ST-LINK debug probe, or a J-Trace debug probe. For J-Trace, the Data Log Summary window is available when ETM trace is disabled. The Data Log Summary window does not display any data when ETM is enabled.
- An SWD interface between the debug probe and the target system.

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

See also *Getting started using data logging*, page 89.

Display area

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

Data*	The name of the data object you have selected to log accesses to. To specify what data object you want to log accesses to, use the Data Log breakpoint dialog box. See <i>Data Log breakpoints dialog box</i> , page 130.
Total accesses**	The number of total accesses.
Read accesses	The number of total read accesses.
Write accesses	The number of total write accesses.

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

Context menu

See *Interrupt Log window context menu*, page 263.

Event Log window

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

Event Log						
Cycles	Program Counter	ITM1	ITM2	ITM3	ITM4	
226458846	---				0x78	
226458868	---			22136		
226458868	---		0x5678			
226462778	---	120				
811029447	0x0000A8BC		0xa8ffe184			
1441029887	0x0000A8E2	2835341700				

Figure 40: Event Log window

To use the Event Log window, you need:

- A J-Link/J-Trace or ST-LINK debug probe
- An SWD interface between the debug probe and the target system.

This window displays event logs produced when the execution passes specific positions in your application code. The Cortex ITM data channels are used for passing the event logs from a running application to the C-SPY Event Log system.

See also *Getting started using event logging*, page 90.

Display area

Each row in the display area shows the event logs in these columns:

Cycles	The number of cycles from the start of the execution until the event. This information is cleared at reset. If a cycle is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it. This column is available when you have selected Show cycles from the context menu.
--------	---

Program Counter*	Displays one of these: An address, which is the content of the PC, that is, the address of the instruction that performed the memory access. ---, the target system failed to provide the debugger with any information. Overflow in red, the communication channel failed to transmit all data from the target system.
ITM1 ITM2 ITM3 ITM4	The Cortex ITM data channels for which the events are logged. For each event log, the event value is displayed. Add a preprocessor macro to your application source code where you want event logs to be generated. See <i>Getting started using event logging</i> , page 90

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

Context menu

See *Interrupt Log window context menu*, page 229.

Event Log Summary window

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

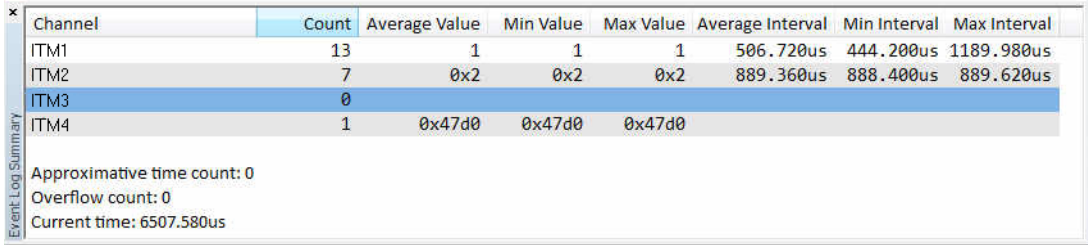


Figure 41: Event Log Summary window

To use the Event Log Summary window, you need:

- A J-Link/J-Trace or ST-LINK debug probe
- An SWD interface between the debug probe and the target system.

This window displays a summary of event logs produced when the execution passes specific positions in your application code. The Cortex ITM data channels are used for passing the event logs from a running application to the C-SPY Event Log system.

See also *Getting started using event logging*, page 90.

Display area

Each row displays the type and the number of accesses to each location in your application code in these columns:

Channel	The name of the communication channel for which event logs are generated.
Count	The number of logged events.
Average Value	The average value of all received event values.
Min Value	The smallest value of all received event values.
Max Value	The largest value of all received event values.
Average Interval	The average time (in cycles) between events.
Min Interval	The shortest time (in cycles) between two events.
Max Interval	The longest time (in cycles) between two events.

At the bottom of the column, the current time or cycles is displayed—execution time since the start of execution or the number of cycles. Overflow count displays the number of overflows. Approximative time count displays the number of inexact timestamps.

Context menu

See *Interrupt Log window context menu*, page 229.

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 111.

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 66.

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.

Data Log breakpoints

Data Log breakpoints are available for the J-Link/J-Trace driver and the ST-LINK driver when you are using a Cortex-M device.

Data Log breakpoints are triggered when data is accessed at the specified location. If you have set a log breakpoint on a specific address or a range, a log message is displayed in the SWO Trace window for each access to that location. A log message can also be displayed in the Data Log window, if that window is enabled. However, these log messages require that you have set up trace data in the SWO Configuration dialog box, see *SWO Configuration dialog box*, page 181.

JTAG watchpoints

The C-SPY J-Link/J-Trace driver and the C-SPY Macraigor driver can take advantage of the JTAG watchpoint mechanism in ARM7/9 cores.

The watchpoints are implemented using the functionality provided by the ARM EmbeddedICE™ macrocell. The macrocell is part of every ARM core that supports the JTAG interface. The EmbeddedICE watchpoint comparator compares the address bus, data bus, CPU control signals and external input signals with the defined watchpoint in real time. When all defined conditions are true, the program will break.

The watchpoints are implicitly used by C-SPY to set code breakpoints or data breakpoints in the application. When setting breakpoints in read/write memory, only one

watchpoint is needed by the debugger. When setting breakpoints in read-only memory, one watchpoint is needed for each breakpoint. Because the macrocell only implements two hardware watchpoints, the maximum number of breakpoints in read-only memory is two.

For a more detailed description of the ARM JTAG watchpoint mechanism, refer to these documents from Advanced RISC Machines Ltd:

- *ARM7TDMI (rev 3) Technical Reference Manual*: chapter 5, *Debug Interface*, and appendix B, *Debug in Depth*
- Application Note 28, *The ARM7TDMI Debug Architecture*.

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:

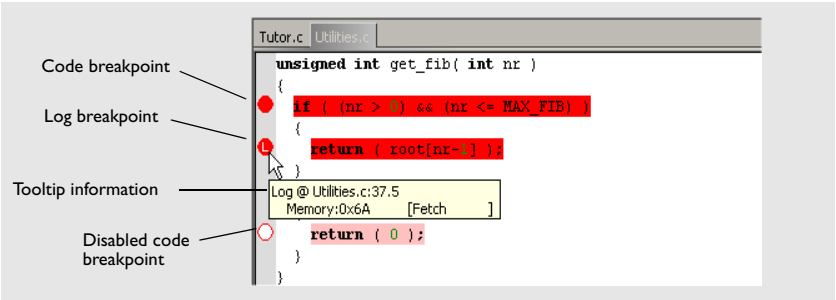


Figure 42: Breakpoint icons



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 for ARM*.



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 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.

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.

For information about the characteristics of breakpoints for the different target systems, see the manufacturer's documentation.

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 **Semihosted** or the **IAR breakpoint** option 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.

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.

BREAKPOINTS OPTIONS

For the following hardware debugger systems it is possible to set some driver-specific breakpoint options before you start C-SPY:

- GDB Server
- J-Link/J-Trace JTAG probes
- Macraigor JTAG probes.

For more information, see *Breakpoints options*, page 132.

BREAKPOINTS ON EXCEPTION VECTORS

You can set breakpoints on exception vectors for ARM9, Cortex-R4, and Cortex-M3 devices. Use the **Vector Catch** dialog box to set a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. For more information, see *Vector Catch dialog box*, page 135.

For the J-Link/J-Trace driver and for RDI drivers, it is also possible to set breakpoints directly on a vector already in the options dialog box, see *Setup options for J-Link/J-Trace*, page 365 and *RDI*, page 377.

SETTING BREAKPOINTS IN __RAMFUNC DECLARED FUNCTIONS

To set a breakpoint in a `__ramfunc` declared function, the program execution must have reached the `main` function. The system startup code moves all `__ramfunc` declared functions from their stored location—normally flash memory—to their RAM location, which means the `__ramfunc` declared functions are not in their proper place and breakpoints cannot be set until you have executed up to the `main` function. Use the **Restore software breakpoints** option to solve this problem, see *Restore software breakpoints at*, page 133.

In addition, breakpoints in `__ramfunc` declared functions added from the editor have to be disabled prior to invoking C-SPY and prior to exiting a debug session.

For information about the `__ramfunc` keyword, see the *IAR C/C++ Development Guide for ARM*.

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.

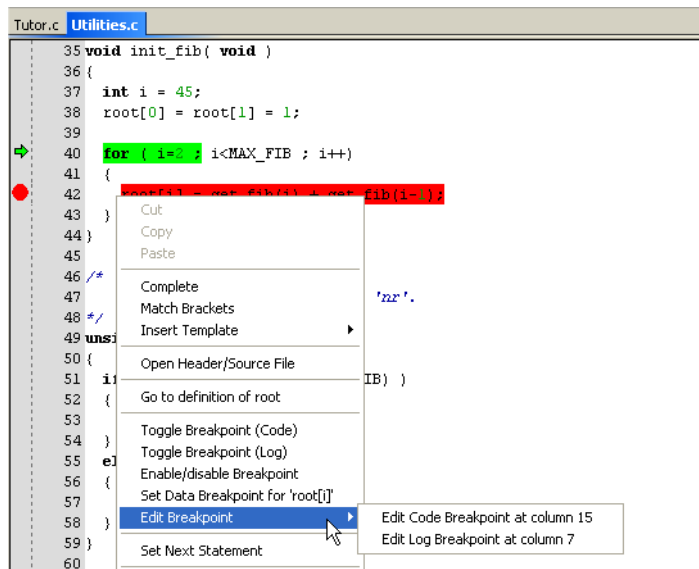


Figure 43: Modifying breakpoints via the context menu

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	J-Link	JTAGjet	RDI	Mac-raigor	GDB Server	ST-LINK	TI Stellaris	TI XDS100	Angel	IAR ROM-monitor
__setCodeBreak	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
__setDataBreak	Yes	No	No	No	No	No	No	No	No	No	No
__setLogBreak	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
__setSimBreak	Yes	No	No	No	No	No	No	No	No	No	No
__setTraceStartBreak	Yes	No	No	No	No	No	No	No	No	No	No
__setTraceStopBreak	Yes	No	No	No	No	No	No	No	No	No	No
__clearBreak	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 10: C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 281.

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 272.

SETTING A BREAKPOINT ON AN EXCEPTION VECTOR

This procedure applies to J-Link/J-Trace and Macraigor.

To set a breakpoint on an exception vector:

- 1 Select the correct device. Before starting C-SPY, choose **Project>Options** and select the **General Options** category. Choose the appropriate core or device from one of the **Processor variant** drop-down lists available on the **Target** page.
- 2 Start C-SPY.
- 3 Choose **J-Link>Vector Catch**. By default, vectors are selected according to your settings on the Breakpoints options page, see *Breakpoints options*, page 132.
- 4 In the **Vector Catch** dialog box, select the vector you want to set a breakpoint on, and click **OK**. The breakpoint will only be triggered at the beginning of the exception.

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 119
- *Breakpoint Usage window*, page 121
- *Code breakpoints dialog box*, page 122

- *JTAG Watchpoints dialog box*, page 124
- *Log breakpoints dialog box*, page 127
- *Data breakpoints dialog box*, page 128
- *Data Log breakpoints dialog box*, page 130
- *Breakpoints options*, page 132
- *Immediate breakpoints dialog box*, page 134
- *Vector Catch dialog box*, page 135
- *Enter Location dialog box*, page 135
- *Resolve Source Ambiguity dialog box*, page 137.

See also:

- *Reference information on C-SPY system macros*, page 281
- *Reference information on trace*, page 176.

Breakpoints window

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

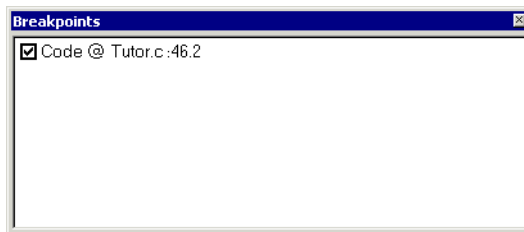


Figure 44: Breakpoints window

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:

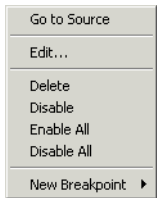


Figure 45: Breakpoints window context menu

These commands are available:

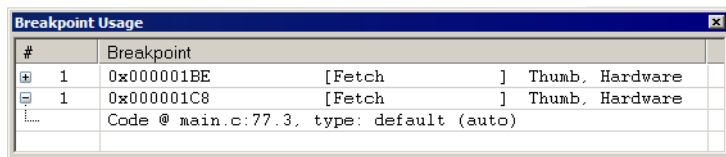
Go to Source	Moves the insertion point to the location of the breakpoint, if the breakpoint has a source location. Double-click a breakpoint in the Breakpoints window to perform the same command.
Edit	Opens the breakpoint dialog box for the breakpoint you selected.
Delete	Deletes the breakpoint. Press 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 window

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



#	Breakpoint
1	0x000001BE [Fetch] Thumb, Hardware
1	0x000001C8 [Fetch] Thumb, Hardware
Code @ main.c:77.3, type: default (auto)	

Figure 46: Breakpoint Usage dialog box

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

The 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** window for:

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

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.

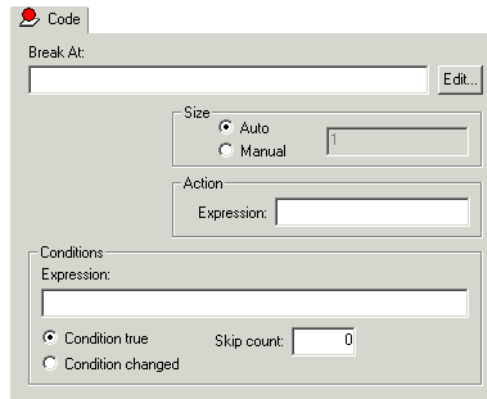


Figure 47: Code breakpoints dialog box

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. For information about support for breakpoints in the C-SPY driver you are using, see *Breakpoints in the C-SPY hardware drivers*, page 111.

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 135.

Breakpoint type

Overrides the default breakpoint type. Select the **Override default** check box and choose between the **Software** and **Hardware** options.

You can specify the breakpoint type for these C-SPY drivers:

- GDB Server
- J-Link/J-Trace JTAG probes
- Macraigor JTAG probes.

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

Determines whether there is an action connected to the breakpoint. 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.

JTAG Watchpoints dialog box

The **JTAG Watchpoints** dialog box is available from the driver-specific menu.

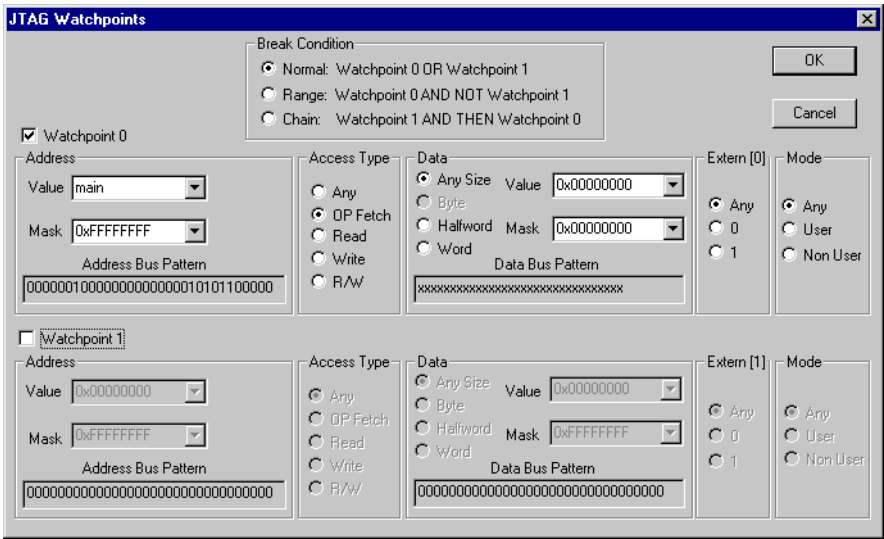


Figure 48: JTAG Watchpoints dialog box

Use this dialog box to directly control the two hardware watchpoint units. If the number of needed watchpoints (including implicit watchpoints used by the breakpoint system) exceeds two, an error message will be displayed when you click the **OK** button. This check is also performed for the C-SPY **Go** button.

This dialog box is available for:

- The J-Link/J-Trace driver
- The Macraigor driver.

To cause a trigger for accesses in the range 0x20-0xFF:

- 1 Set **Break Condition** to **Range**.
- 2 Set the address value of watchpoint 0 to 0 and the mask to 0xFF.
- 3 Set the address value of watchpoint 1 to 0 and the mask to 0x1F.

Address

Specify the address to watch for.

Value	Specify an address or a C-SPY expression that evaluates to an address. Alternatively, you can select an address you have previously watched for from the drop-down list. For detailed information about C-SPY expressions, see <i>C-SPY expressions</i> , page 84.
Mask	Qualifies each bit in the value. A zero bit in the mask will cause the corresponding bit in the value to be ignored in the comparison. To match any address, enter 0. Note that the mask values are inverted with respect to the notation used in the ARM hardware manuals.
Address Bus Pattern	Shows the bit pattern to be used by the address comparator. Ignored bits as specified in the mask are shown as x.

Access Type

Selects the access type of the data to watch for:

Any	Matches any access type.
OP Fetch	Matches an operation code (instruction) fetch.
Read	Reads from location.
Write	Writes to location.
R/W	Reads from or writes to location.

Data

Specifies the data to watch for. For size, choose between:

Any Size	Matches data accesses of any size.
Byte	Matches byte size accesses.
Halfword	Matches halfword size accesses.
Word	Matches word size accesses.

You can specify a value to watch for. Choose between:

Value	Specify a value or a C-SPY expression. Alternatively, you can select a value you have previously watched for from the drop-down list. For detailed information about C-SPY expressions, see <i>C-SPY expressions</i> , page 84.
Mask	Qualifies each bit in the value. A zero bit in the mask will cause the corresponding bit in the value to be ignored in the comparison. To match any address, enter 0. Note that the mask values are inverted with respect to the notation used in the ARM hardware manuals.
Data Bus Pattern	Shows the bit pattern to be used by the address comparator. Ignored bits as specified in the mask are shown as x.

Extern

Defines the state of the external input. Choose between:

Any	Ignores the state.
0	Defines the state as low.
1	Defines the state as high.

Mode

Selects which CPU mode that must be active for a match. Choose between:

User	Selects the CPU mode USER.
Non User	Selects one of the CPU modes SYSTEM SVC, UND, ABORT, IRQ, or FIQ.
Any	Ignores the CPU mode.

Break Condition

Selects how the defined watchpoints will be used. Choose between:

Normal	Uses the two watchpoints individually (OR).
---------------	---

Range	Combines both watchpoints to cover a range where watchpoint 0 defines the start of the range and watchpoint 1 the end of the range. Selectable ranges are restricted to being powers of 2.
Chain	Makes a trigger of watchpoint 1 arm watchpoint 0. A program break will then occur when watchpoint 0 is triggered.

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.

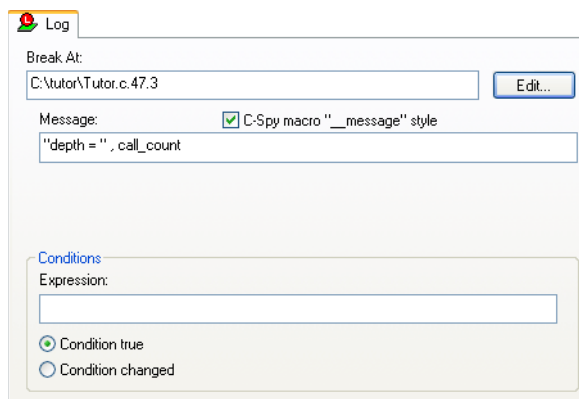


Figure 49: Log breakpoints dialog box

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 111.

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 135.

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 278.

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.

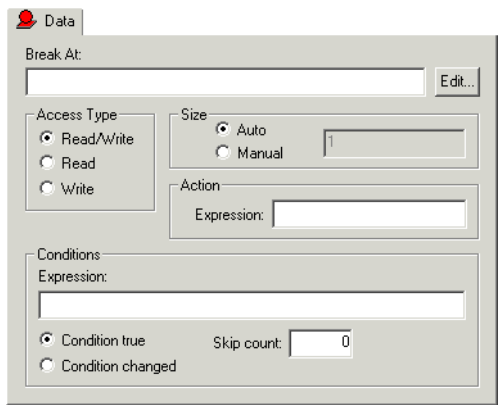


Figure 50: Data breakpoints dialog box

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. For information about support for breakpoints in the C-SPY driver you are using, see *Breakpoints in the C-SPY hardware drivers*, page 111.

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 135.

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.

Trigger range

Shows the requested range and the effective range to be covered by the trace. The range suggested is either within or exactly the area specified by the **Break At** and the **Size** options.

Extend to cover requested range	Extends the breakpoint so that a whole data structure is covered. For data structures that do not fit the size of the possible breakpoint ranges supplied by the hardware breakpoint unit, for example three bytes, the breakpoint range will not cover the whole data structure. Note that the breakpoint range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data.
--	---

Match data

Enables matching of the accessed data. Use the **Match data** options in combination with the access types for data. This option can be useful when you want a trigger when a variable has a certain value.

- Value**
- Specify a data value.
- Mask**
- Specify which part of the value to match (word, halfword, or byte).

The **Match data** options are only available for J-Link/J-Trace and ST-LINK, and when using an ARM7/9 or a Cortex-M device.

Note: For Cortex-M devices, only one breakpoint with Match data can be set. Such a breakpoint uses two hardware breakpoints.

Data Log breakpoints dialog box

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

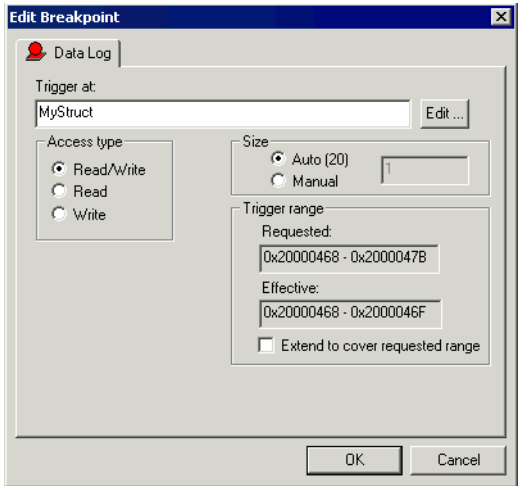


Figure 51: Data Log breakpoints dialog box

Data Log breakpoints are triggered when data is accessed at the specified location. If you have set a log breakpoint on a specific address or a range, a log message is displayed in the SWO Trace window for each access to that location. A log message can also be displayed in the Data Log window, if that window is enabled. Data logs can also be displayed on the Data Log graph in the Timeline window, if that window is enabled. However, these log messages require that you have set up trace data in the **SWO Configuration** dialog box, see *SWO Configuration dialog box*, page 181.

Note: Setting **Data Log** breakpoints is possible only for Cortex-M with SWO using the J-Link debug probe.

Trigger at

Specify the location for the breakpoint in the **Trigger at** text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box; see *Enter Location dialog box*, page 135.

Access Type

Selects the type of memory access that triggers data breakpoints:

Read/Write	Reads from or writes to location.
Read	Reads from location; for Cortex-M3, revision 2 devices only.
Write	Writes to location; for Cortex-M3, revision 2 devices only.

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. This can be useful if Trigger at contains a variable.
Manual	Specify the size of the breakpoint range in the text box.

Trigger range

Shows the requested range and the effective range to be covered by the trace. The range suggested is either within or exactly the area specified by the **Trigger at** and the **Size** options.

Extend to cover requested range

Extends the breakpoint so that a whole data structure is covered. For data structures that do not fit the size of the possible breakpoint ranges supplied by the hardware breakpoint unit, for example three bytes, the breakpoint range will not cover the whole data structure. Note that the breakpoint range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data.

Breakpoints options

The **Breakpoints** option page is available in the **Options** dialog box. Choose **Project>Options**, select the category specific to the debugger system you are using, and click the **Breakpoints** tab.

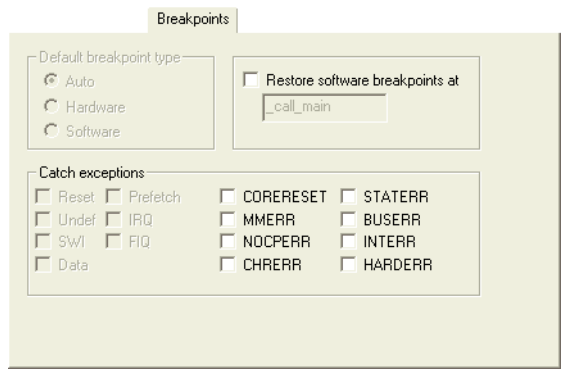


Figure 52: Breakpoints options

For the following hardware debugger systems it is possible to set some driver-specific breakpoint options before you start C-SPY:

- GDB Server
- J-Link/J-Trace JTAG probes
- Macraigor JTAG probes.

Default breakpoint type

Selects the type of breakpoint resource to be used when setting a breakpoint. Choose between:

Auto	Uses a software breakpoint. If this is not possible, a hardware breakpoint will be used. The debugger will use read/write sequences to test for RAM; in that case, a software breakpoint will be used. The Auto option works for most applications. However, there are cases when the performed read/write sequence will make the flash memory malfunction. In that case, use the Hardware option.
Hardware	Uses hardware breakpoints. If it is not possible, no breakpoint will be set.
Software	Uses software breakpoints. If it is not possible, no breakpoint will be set.

Restore software breakpoints at

Automatically restores any breakpoints that were destroyed during system startup.

This can be useful if you have an application that is copied to RAM during startup and is then executing in RAM. This can, for example, be the case if you use the `initialize` by `copy` linker directive for code in the linker configuration file or if you have any `__ramfunc` declared functions in your application.

In this case, all breakpoints will be destroyed during the RAM copying when the C-SPY debugger starts. By using the **Restore software breakpoints at** option, C-SPY will restore the destroyed breakpoints.

Use the text field to specify the location in your application at which point you want C-SPY to restore the breakpoints. The default location is the label `-call_main`.

Catch exceptions

Sets a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. This option is available for ARM9, Cortex-R4, and Cortex-M3 devices. The settings you make will work as default settings for the project. However, you can override these default settings during the debug session by using the **Vector Catch** dialog box, see *Breakpoints on exception vectors*, page 112.

The settings you make will be preserved during debug sessions.

This option is supported by the C-SPY J-Link/J-Trace driver only.

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.

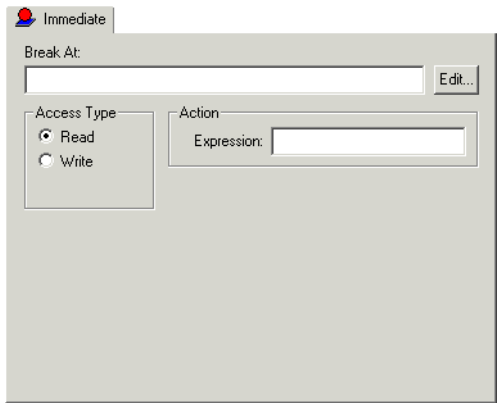


Figure 53: Immediate breakpoints dialog box

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 135.

Access Type

Selects the type of memory access that triggers immediate breakpoints:

Read	Reads from location.
Write	Writes to location.

Action

Determines whether there is an action connected to the breakpoint. Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

Vector Catch dialog box

The **Vector Catch** dialog box is available from the **J-Link** menu for J-Link/J-Trace and Macraigor.

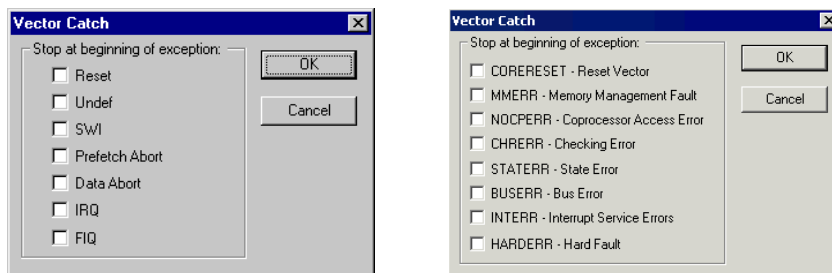


Figure 54: The Vector Catch dialog box—for ARM9/Cortex-R4 versus for Cortex-M3

Use this dialog box to set a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. You can set breakpoints on vectors for ARM9, Cortex-R4, and Cortex-M3 devices. Note that the settings you make here will not be preserved between debug sessions.

Note: For the J-Link/J-Trace driver and for RDI drivers, it is also possible to set breakpoints directly on a vector already in the options dialog box, see *Setup options for J-Link/J-Trace*, page 365 and *RDI*, page 377.

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.

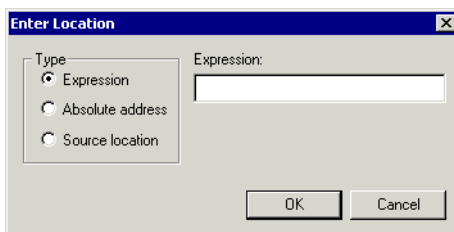


Figure 55: Enter Location dialog box

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	<p>A C-SPY expression, whose value evaluates to a valid address, such as a function or variable name.</p> <p>Code breakpoints are set on functions, for example <code>main</code>. 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 location of the third element of the array <code>arr</code>.</p> <p>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.</p> <p>For more information about C-SPY expressions, see <i>C-SPY expressions</i>, page 84.</p>
Absolute address	<p>An absolute location on the form <code>zone:hexaddress</code> or simply <code>hexaddress</code> (for example <code>Memory:0x42</code>). <code>zone</code> refers to C-SPY memory zones and specifies in which memory the address belongs.</p>
Source location	<p>A location in your C source code using the syntax: <code>{filename}.row.column</code>.</p> <p><code>filename</code> specifies the filename and full path. <code>row</code> specifies the row in which you want the breakpoint. <code>column</code> specifies the column in which you want the breakpoint.</p> <p>For example, <code>{C:\src\prog.c}.22.3</code> sets a breakpoint on the third character position on line 22 in the source file <code>Utilities.c</code>.</p> <p>Note that the Source location type is usually meaningful only for code breakpoints.</p>

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.

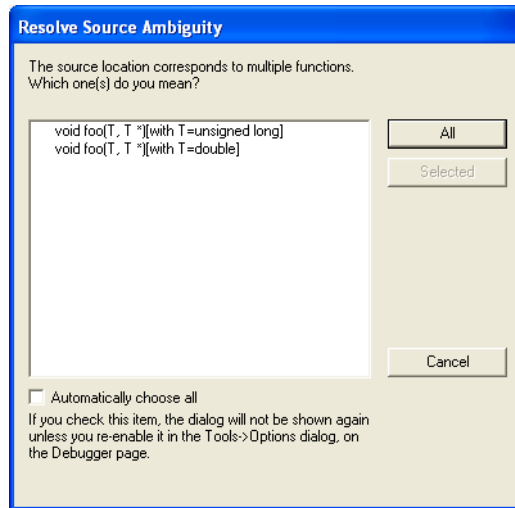


Figure 56: Resolve Source Ambiguity dialog box

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 for ARM*.

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
- Memory access checking.

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. Except for the hardwired group of CPU registers, additional registers are defined in the device description file. These registers are the device-specific memory-mapped control and status registers for the peripheral units on the ARM devices.

Due to the large amount of registers, it is inconvenient to show all registers concurrently in the Register window. Instead you can divide registers into *register groups*. The device description file defines one group for each peripheral unit in the device. You can also define your own groups by choosing **Tools>Options>Register Filter**. 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. The ARM architecture has only one zone, `Memory`, which covers the whole ARM memory range.

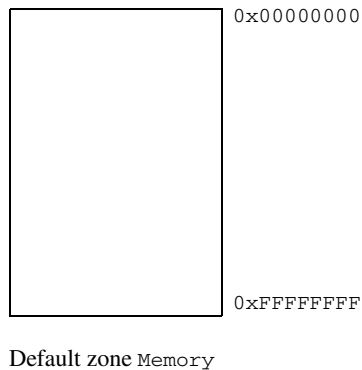


Figure 57: Zones in C-SPY

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.

These zones are available depending on the device description file you are using: `Memory`, `Memory8`, `Memory16`, `Memory32`, and `Memory64`.

For normal memory, the default zone `Memory` can be used, but certain I/O registers might require to be accessed as 8, 16, 32, or 64 bits to give correct results. By using different memory zones, you can control the access width used for reading and writing in, for example, the Memory window.

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 cores 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 section 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 section 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++ Development Guide for ARM*.

MEMORY ACCESS CHECKING

The C-SPY simulator can simulate various memory access types of the target hardware and detect illegal accesses, for example a read access to write-only memory. If a memory access occurs that does not agree with the access type specified for the specific memory area, C-SPY will regard this as an illegal access. Also, a memory access to memory which is not defined is regarded as an illegal access. The purpose of memory access checking is to help you to identify any memory access violations.

The memory areas can either be the zones predefined in the device description file, or memory areas based on the section information available in the debug file. In addition to these, you can define your own memory areas. The access type can be read and write, read-only, or write-only. You cannot map two different access types to the same memory area. You can check for access type violation and accesses to unspecified ranges. Any violations are logged in the Debug Log window. You can also choose to have the execution halted.

Reference information on memory and registers

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

- *Memory window*, page 143
- *Memory Save dialog box*, page 147
- *Memory Restore dialog box*, page 148
- *Fill dialog box*, page 148
- *Symbolic Memory window*, page 150
- *Stack window*, page 152
- *Register window*, page 155
- *Memory Access Setup dialog box*, page 157
- *Edit Memory Access dialog box*, page 159.

Memory window

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

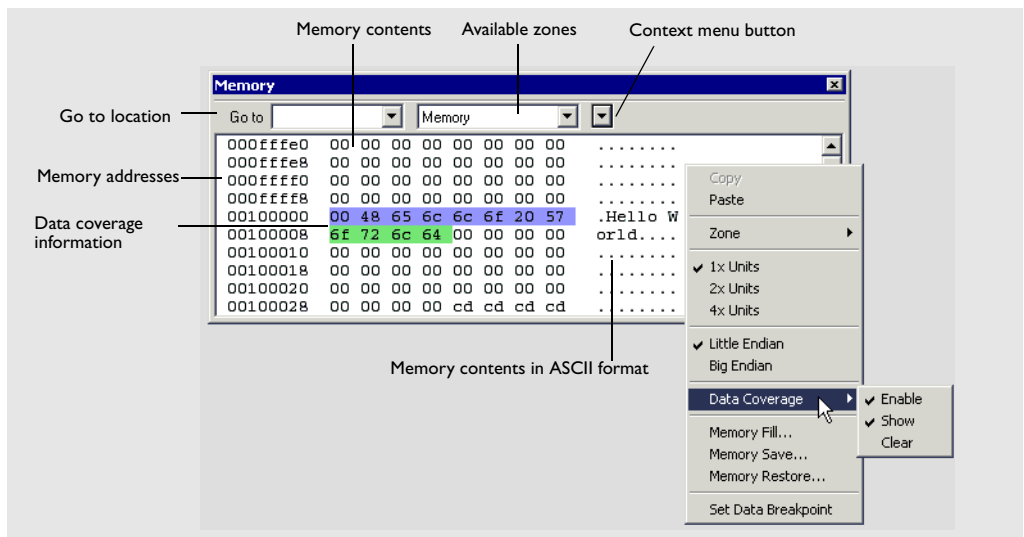


Figure 58: Memory window

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 141.
Context menu button	Displays the context menu, see <i>Context menu</i> , page 145.
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:

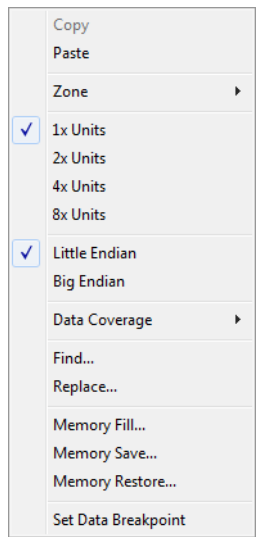


Figure 59: Memory window context menu

These commands are available:

Copy, Paste	Standard editing commands.
Zone	Selects a memory zone to display, see <i>C-SPY memory zones</i> , page 141.
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.
8x Units	Displays the memory contents in units of 64 bits.
Little Endian	Displays the contents in little-endian byte order.
Big Endian	Displays the contents in big-endian byte order.

Data Coverage	<p>Choose between:</p> <p>Enable toggles data coverage on or off.</p> <p>Show toggles between showing or hiding data coverage.</p> <p>Clear clears all data coverage information.</p> <p>These commands are only available if your C-SPY driver supports data coverage.</p>
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 for ARM</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 for ARM</i> .
Memory Fill	Displays a dialog box, where you can fill a specified area with a value, see <i>Fill dialog box</i> , page 148.
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 147.
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 148.
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 115.

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.

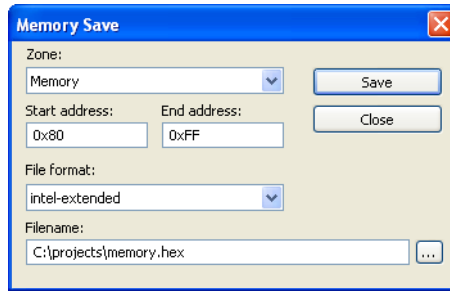


Figure 60: Memory Save dialog box

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.

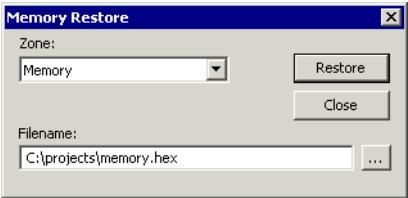


Figure 61: Memory Restore dialog box

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.

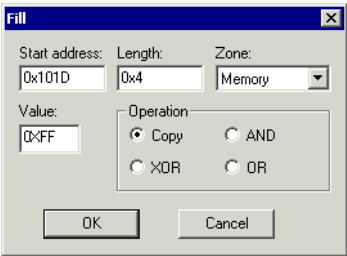


Figure 62: Fill dialog box

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.

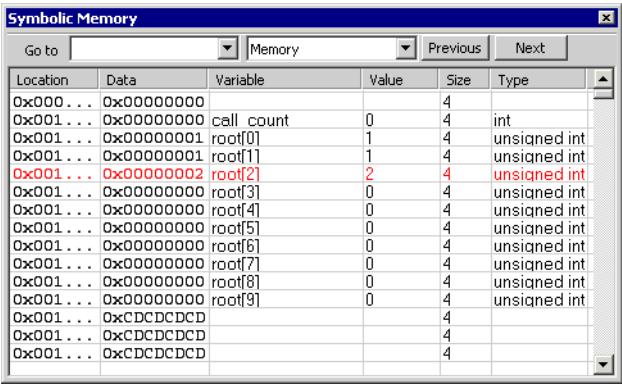


Figure 63: Symbolic Memory window

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 *C-SPY memory zones*, page 141.
- 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:

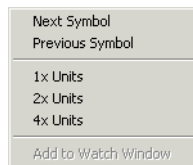


Figure 64: Symbolic Memory window context menu

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.

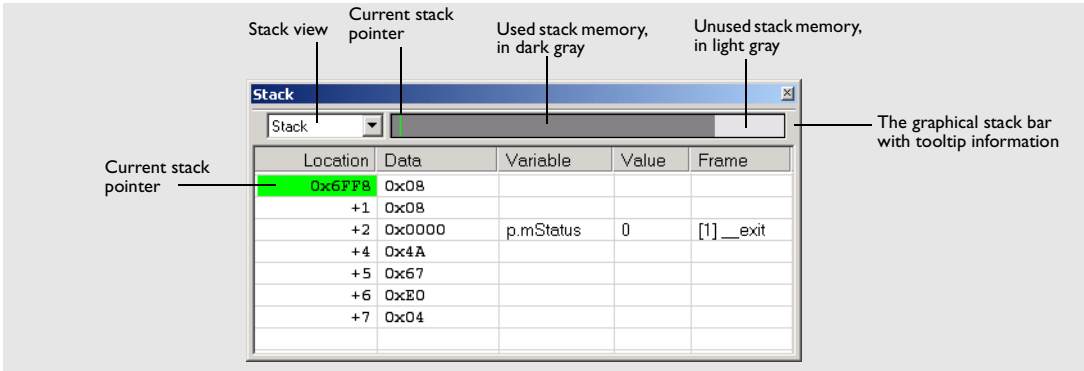


Figure 65: Stack window

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.

Overriding the default stack setup

The Stack window retrieves information about the stack size and placement from the definition of the sections holding the stacks made in the linker configuration file. The sections are described in the *IAR C/C++ Development Guide for ARM*.

For applications that set up the stacks using other mechanisms, it is possible to override the default mechanism. Use one of the C-SPY command line option variants, see *--proc_stack_stack*, page 342.

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 111.

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

Toolbar

Stack	Selects which stack to view. This applies to cores 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:

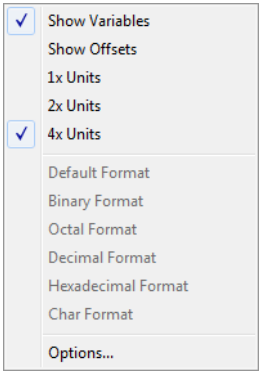


Figure 66: Stack window context menu

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 8, <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 for ARM</i> .

Register window

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

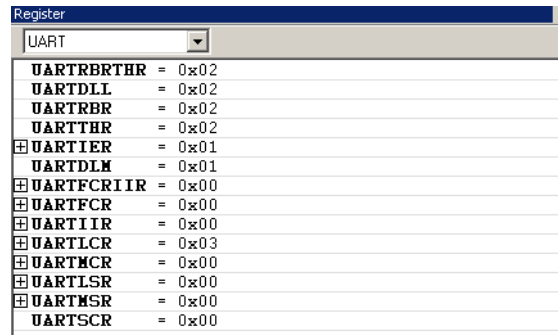


Figure 67: Register window

This window gives an up-to-date display of the contents of the processor registers and special function registers, and allows you to edit their contents. Optionally, you can choose to load either predefined register groups or to define your own application-specific groups

You can open several instances of this window, which is very convenient if you want to keep track of different register groups.

To enable predefined register groups:

- 1 Select a device description file that suits your device, see *Selecting a device description file*, page 47.
- 2 The register groups appear in the Register window, provided that they are defined in the device description file. Note that the available register groups are also listed on the **Register Filter** page.

To define application-specific register groups, read about register filter options in the *IDE Project Management and Building Guide for ARM*.

Toolbar

Drop-down list

Selects which register group to display, by default CPU Registers. By default, there are two register groups in the debugger:

Current CPU Registers contains the registers that are available in the current processor mode.

CPU Registers contains both the current registers and their banked counterparts available in other processor modes.

Additional register groups are predefined in the device description files—available in the `arm\config` directory—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**.

Memory Access Setup dialog box

The **Memory Access Setup** dialog box is available from the **Simulator** menu.

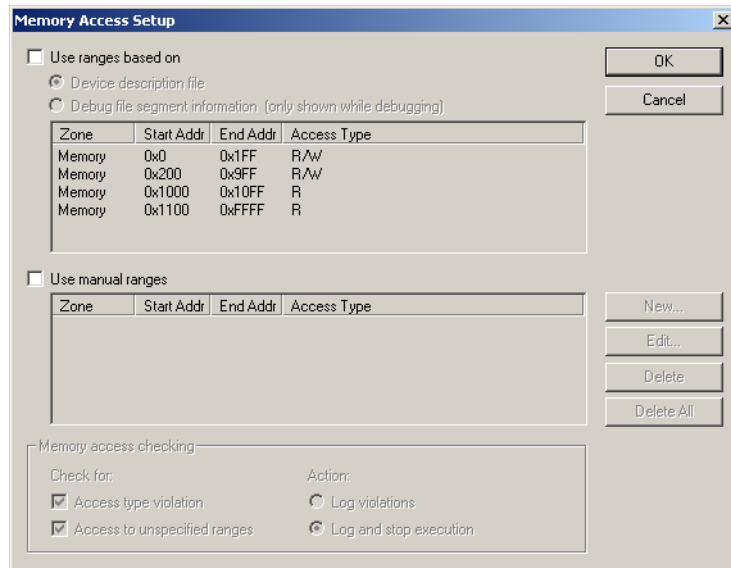


Figure 68: Memory Access Setup dialog box

This dialog box lists all defined memory areas, where each column in the list specifies the properties of the area. In other words, the dialog box displays the memory access setup that will be used during the simulation.

Note: If you enable both the **Use ranges based on** and the **Use manual ranges** option, memory accesses are checked for all defined ranges.

For information about the columns and the properties displayed, see *Edit Memory Access dialog box*, page 159.

Use ranges based on

Selects any of the predefined alternatives for the memory access setup. Choose between:

Device description file Loads properties from the device description file.

Debug file segment information	Properties are based on the section information available in the debug file. This information is only available while debugging. The advantage of using this option, is that the simulator can catch memory accesses outside the linked application.
---------------------------------------	--

Use manual ranges

Specify your own ranges manually via the **Edit Memory Access** dialog box. To open this dialog box, choose **New** to specify a new memory range, or select a memory zone and choose **Edit** to modify it. For more information, see *Edit Memory Access dialog box*, page 159.

The ranges you define manually are saved between debug sessions.

Memory access checking

Check for determines what to check for;

- Access type violation
- Access to unspecified ranges.

Action selects the action to be performed if an access violation occurs; choose between:

- Log violations
- Log and stop execution.

Any violations are logged in the Debug Log window.

Buttons

These buttons are available:

New	Opens the Edit Memory Access dialog box, where you can specify a new memory range and attach an access type to it, see <i>Edit Memory Access dialog box</i> , page 159.
Edit	Opens the Edit Memory Access dialog box, where you can edit the selected memory area. See <i>Edit Memory Access dialog box</i> , page 159.
Delete	Deletes the selected memory area definition.
Delete All	Deletes all defined memory area definitions.

Note: Except for the OK and Cancel buttons, buttons are only available when the option **Use manual ranges** is selected.

Edit Memory Access dialog box

The **Edit Memory Access** dialog box is available from the **Memory Access Setup** dialog box.

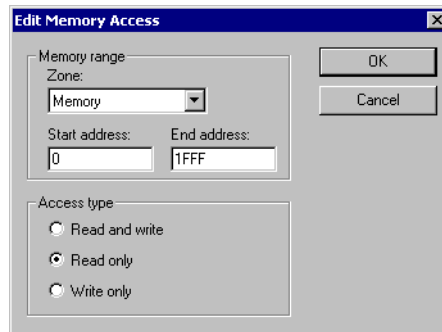


Figure 69: Edit Memory Access dialog box

Use this dialog box to specify the memory ranges, and assign an access type to each memory range, for which you want to detect illegal accesses during the simulation.

Memory range

Defines the memory area for which you want to check the memory accesses:

Zone	Selects a memory zone, see <i>C-SPY memory zones</i> , page 141.
Start address	Specify the start address for the address range, in hexadecimal notation.
End address	Specify the end address for the address range, in hexadecimal notation.

Access type

Selects an access type to the memory range; choose between:

- **Read and write**
- **Read only**
- **Write only.**

Collecting and using trace data in the JTAGjet driver

This chapter gives you information about using the JTAGjet trace.

Using JTAGjet trace

This section gives information about collecting and using trace data. More specifically, you will get information about:

- *Briefly about using JTAGjet trace*, page 161
- *The JTAGjet Trace window*, page 162
- *Trace view field configuration dialog box*, page 164
- *Trace search query dialog box*, page 165
- *ETM Control dialog box*, page 167
- *ETM Configuration dialog box*, page 168.

BRIEFLY ABOUT USING JTAGJET TRACE

The trace buffer contains a chronologically ordered collection of processor state “snapshots” from the past. In its most common form, it is a contiguous collection of processor bus cycles. It might, however, consist of multiple snippets as short as a single sample taken over a longer period of time. The trace buffer width (the number of bits of information contained in every sample) and depth (the number of samples that can be stored in the buffer) vary with different targets. For example, some emulators have trace buffers 120 bits wide and 32K samples deep. The Trace window allows you to use this information, by providing you with powerful customization, filtration and search capabilities.

Only if the target features a trace buffer, such as that found in in-circuit emulators, or it has a trace FIFO, will the window appear. The Trace window provides access to the execution trace buffer on the target board.

The right side of the status bar at the bottom of the window shows the Trace Clock frequency.

The JTAGjet-Trace hardware measures the frequency of the trace clock even when the trace is not being collected. Trace Clock with a value of 0 indicates that the trace clock is not available. Re-enable the trace by clicking the **Enable** button, as instructed by the Trace window.

The JTAGjet Trace window

The JTAGjet Trace window is available from the JTAGjet menu.

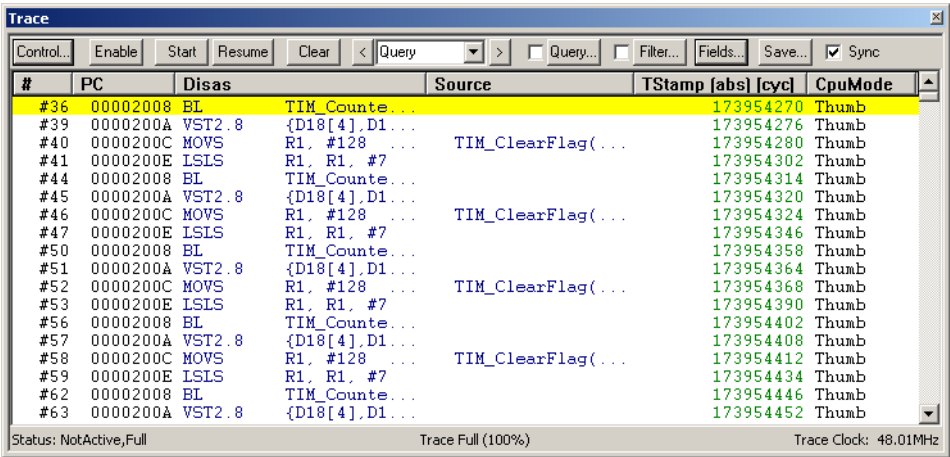


Figure 70: The JTAGjet Trace window

Field format selection

You can control the format in which each individual column in the Trace window is displayed. To do so, click on the header of the column you would like to change and select a format from the menu. For numerical fields, the display formats define the radix used for displaying the value. For timestamp fields, a number of other options described below are available.

Timestamp display

Some target systems, such as in-circuit emulators, are capable of inserting a timestamp in each trace sample. The timestamp can be used to determine the exact duration of certain events and to analyze the dynamics of the system's behavior. The Trace window offers several display formats to aid you in timing analysis. From the column's drop-down menu, select the units in which you want to measure time, such as seconds, milliseconds, microseconds or nanoseconds. Time intervals can be displayed as **Absolute** (the time contained in the sample is displayed as is), **Relative** (the time of a sample determined after subtracting the 'base' time from each sample), or **Delta** (the timestamp difference between two consecutive samples).

Trace filtration

The trace window allows you to filter the trace buffer to show only samples of interest. The trace filtration process is very similar to the search process, except that a trace query is not used to find a sample, but to decide whether to display it or not. The trace filter is configured using a trace query specification identical to the one used for trace searching. For more information, see *Trace search query dialog box*, page 165.

Once a filter query has been defined, you can turn filtration on and off by clicking on the **Filter** button on the trace window toolbar. When a trace filter is active (ON), only samples satisfying the filter query are displayed. The positions of samples that were filtered out are indicated by dashed red lines in the trace window.

Saving trace

The **Save** button, as well as the corresponding context menu command, allows you to save the trace to a file in one of these formats:

Format	Description
==>	Add selected field on end of visible fields
CSV	Can be opened in Microsoft Excel or a text editor. Cannot be opened in the Trace window.
TID	Text Trace Test Data file that is reloadable. Can be opened in a text editor or used by external programs.
TDF	Binary Trace Data File. Not supported.

Table 11: JTAGjet trace save formats

You can choose to save:

Visible records	Saves only samples shown on the screen.
All records	Caution is advised: The option might create very large files.
Range of records	Saves the samples between the selected starting and ending samples.

Saving the trace as a text file preserves only the fields actually displayed on the screen. The **Field** option allows you to exclude certain fields.

The save operation takes into account the current filtering settings; only those samples that meet the filtering conditions are saved.

Trace view field configuration dialog box

The Trace window allows you to select and rearrange the fields of the collected trace information. To configure these fields, click the **Fields** button on the Trace window toolbar. The system will display the **Trace view field configuration** dialog box.

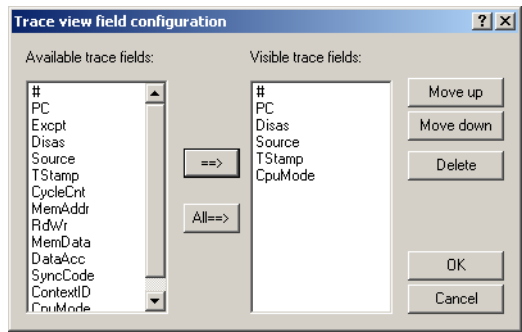


Figure 71: Trace view field configuration dialog box

You can display a field in multiple copies or completely remove it. Multiple copies of the same field might be useful, for example, when you want to see the same information in different formats, such as hex, decimal and binary, at the same time.

These fields are available:

#	Frame sequence number. The smaller #, the older sample.
PC	Program Counter
Disas	Disassembled code (in case there is no source file)
Source	Source code associated with the trace frame
MemAddr	Address of memory variable being read or written
RdWr	Data memory operation (Rd or Wr)
MemData	Variable value being read or written
TStamp	Timestamp
DataAcc	Type of memory access (Byte, Word, Half)

- SyncCode** Trace status, one of:
- Start, start of trace after CPU stop
 - Sync, ETM sync frames
 - On, start of trace after each trigger
 - FIFO, ETM FIFO had an overflow (loss of trace data)
- CpuMode** CPU execution mode, one of:
- ARM, ARM mode execution
 - Thumb, Thumb mode execution
 - NoExec, Instruction fetched but not executed

Note: The available trace fields might be different for some CPUs.

Trace search query dialog box

The JTAGjet Trace window allows you to search the trace buffer using sophisticated queries.

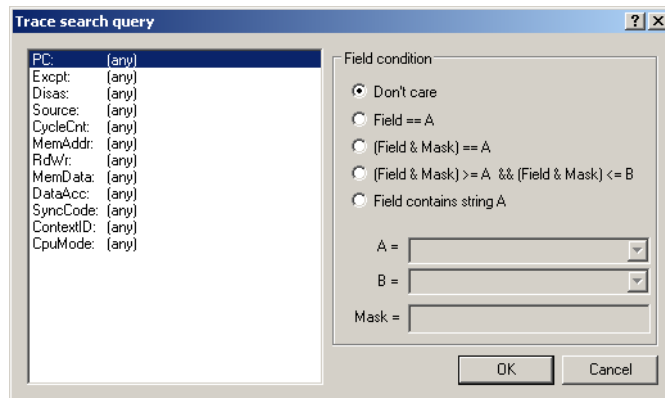


Figure 72: Trace search query dialog box

Trace queries can range from simple, such as “sample with address bus value equal to 0x1234” to very complicated, such as “sample containing a fetch cycle of an instruction at an odd address in the upper 16 Kbytes of memory whose operand contained the word bptr”. Both of these queries can be reduced to sets of conditions that each field in a sample has to satisfy. You can specify the field conditions using the trace search query dialog box accessed by clicking the **Query** button on the Trace window toolbar.

For every field in the trace sample displayed in the list box on the left, you can define a condition that the field value must satisfy. This condition is specified in the **Field condition** section of the dialog box. Choose between these conditions:

Condition	Description
Don't care	Any value is acceptable.
Field == A	The field must have the value specified in text box A .
(Field & Mask) == A	The field value bitwise AND-ed with the value in the Mask box must equal to the value specified in text box A .
(Field & Mask) >= A && (Field & Mask) <= B	The field value bitwise AND-ed with the value in the Mask box must be greater then, or equal to, the value specified in text box A , and less than or equal to the value specified in text box B .
Field contains string A	The textual representation of the field value must contain the string specified in text box A .

Table 12: JTAGjet trace searching conditions

Once you have defined a trace query, you can employ it to navigate through the trace buffer using the **Find next** and **Find prev.** buttons on the Trace window toolbar. You can also tag (highlight) all samples satisfying the current query by clicking the **Tag** button. The label of the button indicates if the tag is set (ON) or not (OFF).

ETM Control dialog box

To display the **ETM Control** dialog box, click the **Control** button in the left corner of the Trace window.

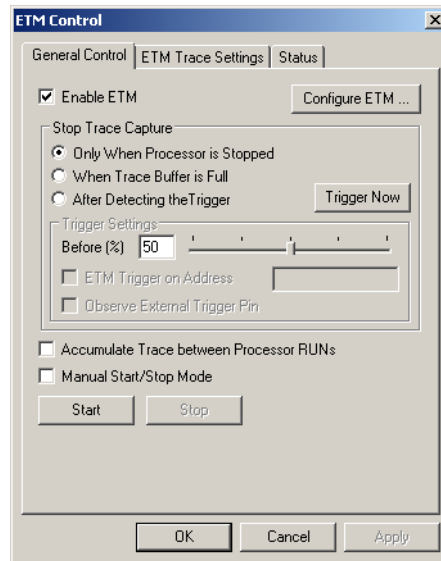


Figure 73: ETM Control dialog box

The **General Control** tab allows you to control the processes of starting and stopping trace capture. Clicking the button **Configure ETM** displays the **ETM Configuration** dialog box, see *ETM Configuration dialog box*, page 168.

The **ETM Trace Settings** tab lets you decide what will be captured in the trace buffer.

The **Status** tab shows the JTAGjet-Trace hardware capabilities and the on-chip ETM resources of the processor in use. These capabilities are determined by the device manufacturer and the ETM architecture.

ETM Configuration dialog box

The **ETM Configuration** dialog box is displayed when you click the **Configure ETM** button in the **ETM Control** dialog box.

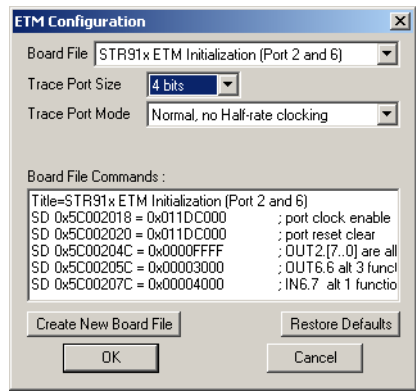


Figure 74: ETM Configuration dialog box

Select an appropriate board file from the **Board File** drop-down list.

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

- *Getting started using data logging*, page 89
- *Debugging in the power domain*, page 225
- *Getting started using interrupt logging using C-SPY hardware drivers*, page 253
- *Using the profiler*, page 213.

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.

Reasons for using the trace triggers and trace filters

By using trace trigger and trace filter conditions, you can select the interesting parts of your source code and use the trace buffer in the J-Trace probe more efficiently. Trace

triggers—Trace Start and Trace Stop breakpoints—specify for example a code section for which you want to collect trace data. A trace filter specifies conditions that, when fulfilled, activate the trace data collection during execution.

For ARM7/9 devices, you can specify up to 16 trace triggers and trace filters in total, of which 8 can be trace filters.

For Cortex-M devices, you can specify up to 4 trace triggers and trace filters in total.

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.

C-SPY supports collecting trace data from these target systems:

- Devices with support for ETM (Embedded Trace Macrocell)—ETM trace
- Devices with support for the SWD (Serial Wire Debug) interface using the SWO (Serial Wire Output) communication channel—SWO trace
- The C-SPY simulator.

Depending on your target system, different types of trace data can be generated.

ETM trace

ETM trace (also known as full trace) is a continuously collected sequence of every executed instruction for a selected portion of the execution. It is only possible to collect as much data as the buffer can hold.

The debug probe contains a trace buffer that collects trace data in real time, but the data is not displayed in the C-SPY windows until after the execution has stopped.

SWO trace

SWO trace is a sequence of events of various kinds, generated by the on-chip debug hardware. The events are transmitted in real time from the target system over the SWO communication channel. This means that the C-SPY windows are continuously updated while the target system is executing. The most important events are:

- PC sampling

The hardware can sample and transmit the value of the program counter at regular intervals. This is not a continuous sequence of executed instructions (like ETM trace), but a sparse regular sampling of the PC. A modern ARM CPU typically executes millions of instructions per second, while the PC sampling rate is usually counted in thousands per second.

- Interrupt logs

The hardware can generate and transmit data related to the execution of interrupts, generating events when entering and leaving an interrupt handler routine.

- Data logs

Using Data Log breakpoints, the hardware can be configured to generate and transmit events whenever a certain variable, or simply an address range, is accessed by the CPU.

The SWO channel has limited throughput, so it is usually not possible to use all the above features at the same time, at least not if either the frequency of PC sampling, of interrupts, or of accesses to the designated variables is high.

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 and triggers to control the collection of trace data.

If you use the C-SPY J-Link/J-Trace driver or the ST-LINK driver, you have access to windows such as the Interrupt Log, Interrupt Log Summary, Data Log, and Data Log Summary windows.



When you are debugging, two buttons labeled **ETM** and **SWO**, respectively, are visible on the IDE main window toolbar. If any of these buttons is green, it means that the corresponding trace hardware is generating trace data. Just point at the button with the mouse pointer to get detailed tooltip information about which C-SPY features that have requested trace data generation. This is useful, for example, if your SWO communication channel often overflows because too many of the C-SPY features are currently using trace data. Clicking on the buttons opens the corresponding setup dialog boxes.

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.

To use trace-related functionality in your hardware debugger system, you need debug components (hardware, a debug probe, and a C-SPY driver) that all support trace.

Note: The specific set of debug components you are using determine which trace features in C-SPY that are supported.

Requirements for using ETM trace

ETM trace is available for some ARM devices.

To use ETM trace you need one of these combinations:

- A J-Trace debug probe and a device that supports ETM. Make sure to use the C-SPY J-Link/J-Trace driver.
- A J-Link debug probe and a device that supports ETM via ETB (Embedded Trace Buffer). The J-Link probe reads ETM data from the ETB buffer. Make sure to use the C-SPY J-Link/J-Trace driver.

Requirements for using SWO trace

To use SWO trace you need a J-Link, J-Trace, or ST-LINK debug probe that supports the SWO communication channel and a device that supports the SWD/SWO interface.

Requirements for using the trace triggers and trace filters

The trace triggering and trace filtering features are available only for J-Trace and when using an ARM7/9 or Cortex-M device.

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
- Getting started with ETM trace
- Trace data collection using breakpoints
- Searching in trace data
- Browsing through trace data.

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 184.

GETTING STARTED WITH ETM TRACE

To get started using ETM trace:

- 1 Before you start C-SPY:
 - For J-Trace no specific settings are required before starting C-SPY
 - For your device, the trace port must be set up. For some devices this is done automatically when the trace logic is enabled. However, for some devices, typically Atmel and ST devices based on ARM 7 or ARM 9, you need to set up the trace port explicitly. You do this by means of a C-SPY macro file. You can find examples of such files (*ETM_init*.mac*) in the example projects. To use a macro file, choose **Project>Options>Debugger>Setup>Use macro files**. Specify your macro file; a browse button is available for your convenience.

Note that the pins used on the hardware for the trace signals cannot be used by your application.
- 2 After you have started C-SPY, choose **Trace Settings** from the C-SPY driver menu. In the **Trace Settings** dialog box that appears, check if you need to change any of the default settings. For more information, see *ETM Trace Settings dialog box*, page 177.
-  3 Open the Trace window—available from the driver-specific menu—and click the **Activate** button to enable trace data collection.
- 4 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 184.

GETTING STARTED WITH SWO TRACE

To get started using SWO trace:

- 1 Before you start C-SPY, choose **Project>Options>J-Link/J-Trace** for J-Link/J-Trace or **Project>Options>ST-Link** for ST-LINK, respectively. Click the **Connection** tab and choose **Interface>SWD**.
- 2 After you have started C-SPY, choose **SWO Trace Windows Settings** from the **J-Link** menu or the **ST-LINK** menu. In the **SWO Trace Windows Settings** dialog box that appears, make your settings for controlling the output in the Trace window. To see statistical trace data, select the option **Force>PC samples**, see *SWO Trace Window Settings dialog box*, page 179.
- 3 To configure the hardware's generation of trace data, click the **SWO Configuration** button available in the **SWO Configuration** dialog box. For more information, see *SWO Configuration dialog box*, page 181.

Note specifically these settings:



- The value of the **CPU clock** option must reflect the frequency of the CPU clock speed at which the application executes. Note also that the settings you make are preserved between debug sessions.
- To decrease the amount of transmissions on the communication channel, you can disable the **Timestamp** option. Alternatively, set a lower rate for PC Sampling or use a higher SWO clock frequency.



- 4 Open the SWO Trace window—available from the **J-Link/J-Trace** menu or the **ST-LINK** menu, respectively—and click the **Activate** button to enable trace data collection.
- 5 Start the execution. The Trace window is continuously updated with trace data. For more information about this window, see *Trace window*, page 184.

SETTING UP CONCURRENT USE OF ETM AND SWO

If you have a J-Trace debug probe for Cortex-M3, you can use ETM trace and SWO trace concurrently.

In this case, if you activate the ETM trace and the SWO trace, SWO trace data will also be collected in the ETM trace buffer, instead of being streamed via the SWO channel. This means that the SWO trace data will not be displayed until the execution has stopped, instead of being continuously updated live in the SWO Trace window.

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**, **Trace Stop**, or **Trace Filter**.
- The C-SPY system macros `__setTraceStartBreak` and `__setTraceStopBreak` can also be used.

For more information about these breakpoints, see *Trace Start breakpoints dialog box (simulator)*, page 198 and *Trace Stop breakpoints dialog box (simulator)*, page 199, respectively.

Using the trace triggers and trace filters:

- 1 Use the **Trace Start** dialog box to set a start condition—a start trigger—to start collecting trace data.
- 2 Use the **Trace Stop** dialog box to set a stop condition—a stop trigger—to stop collecting trace data.

- 3 Optionally, set additional conditions for the trace data collection to continue. Then set one or more trace filters, using the **Trace Filter** dialog box.
- 4 If needed, set additional trace start or trace stop conditions.
- 5 Enable the Trace window and start the execution.
- 6 Stop the execution.
- 7 You can view the trace data in the Trace window and in browse mode also in the Disassembly window, where also the trace marks for your trace triggers and trace filters are visible.
- 8 If you have set a trace filter, the trace data collection is performed while the condition is true plus some further instructions. When viewing the trace data and looking for a certain data access, remember that the access took place one instruction earlier.

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 209.

- 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 210.

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:

- *ETM Trace Settings dialog box*, page 177
- *SWO Trace Window Settings dialog box*, page 179
- *SWO Configuration dialog box*, page 181
- *Trace window*, page 184
- *Trace Save dialog box*, page 188
- *Function Trace window*, page 189
- *Timeline window*, page 190
- *Power Log window*, page 236
- *Trace Start breakpoints dialog box (simulator)*, page 198
- *Trace Stop breakpoints dialog box (simulator)*, page 199
- *Trace Start breakpoints dialog box*, page 200
- *Trace Stop breakpoints dialog box*, page 203
- *Trace Filter breakpoints dialog box*, page 205
- *Trace Expressions window*, page 208
- *Find in Trace dialog box*, page 209
- *Find in Trace window*, page 210.

ETM Trace Settings dialog box

The ETM Trace Settings dialog box is available from the C-SPY driver menu.

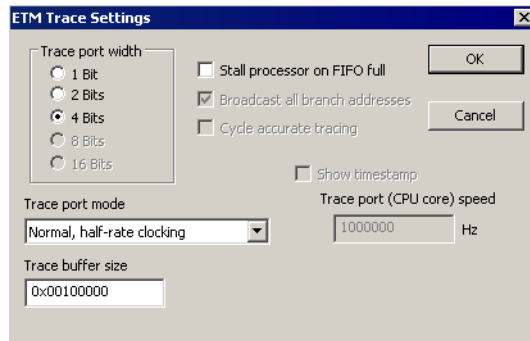


Figure 75: ETM Trace Settings dialog box

This dialog box is available for the:

- J-Link/J-Trace driver

Use this dialog box to configure ETM trace generation and collection.

See also:

- *Requirements for using ETM trace*, page 172
- *Getting started with ETM trace*, page 173.

Trace port width

Specifies the trace bus width, which can be set to 1, 2, 4, 8, or 16 bits. The value must correspond with what is supported by the hardware and the debug probe. For Cortex-M3, 1, 2, and 4 bits are supported by the J-Trace debug probe. For ARM7/9, only 4 bits are supported by the J-Trace debug probe.

Trace port mode

Specifies the used trace clock rate:

- Normal, full-rate clocking
- Normal, half-rate clocking
- Multiplexed
- Demultiplexed
- Demultiplexed, half-rate clocking.

Note: For RDI drivers, only the two first alternatives are available. For the J-Trace driver, the available alternatives depend on the device you are using.

Trace buffer size

Specify the size of the trace buffer. By default, the number of trace frames is 0xFFFF. For ARM7/9 the maximum number is 0xFFFFFF, and for Cortex-M3 the maximum number is 0x3FFFFFF.

For ARM7/9, one trace frame corresponds to 2 bytes of the physical J-Trace buffer size. For Cortex-M3, one trace frame corresponds to approximately 1 byte of the buffer size.

Note: The **Trace buffer size** option is only available for the J-Trace driver.

Cycle accurate tracing

Emits trace frames synchronous to the processor clock even when no trace data is available. This makes it possible to use the trace data for real-time timing calculations. However, if you select this option, the risk for FIFO buffer overflow increases.

Note: This option is only available for ARM7/9 devices.

Broadcast all branches

Makes the processor send more detailed address trace information. However, if you select this option, the risk for FIFO buffer overflow increases.

Note: This option is only available for ARM7/9 devices. For Cortex, this option is always enabled.

Stall processor on FIFO full

Stalls the processor in case the FIFO buffer fills up. The trace FIFO buffer might in some situations become full—FIFO buffer overflow—which means trace data will be lost.

Show timestamp

Makes the Trace window display seconds instead of cycles in the **Index** column. To make this possible you must also specify the appropriate speed for your CPU in the **Trace port (CPU core) speed** text box.

Note: This option is only available when you use the J-Trace driver with ARM7/9 devices.

SWO Trace Window Settings dialog box

The **SWO Trace Window Settings** dialog box is available from the **J-Link** menu or the **ST-LINK** menu, respectively, alternatively from the SWO Trace window toolbar.

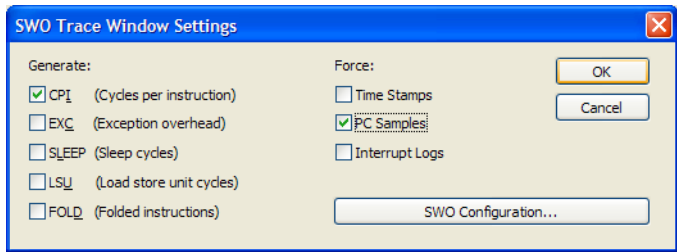


Figure 76: SWO Trace Window Settings dialog box

Use this dialog box to specify what to display in the SWO Trace window.

Note that you also need to configure the generation of trace data, click **SWO Configuration**. For more information, see *SWO Configuration dialog box*, page 181.

Force

Enables data generation, if it is not already enabled by other features using SWO trace data. The Trace window displays all generated SWO data. Other features in C-SPY, for example Profiling, can also enable SWO trace data generation. If no other feature has enabled the generation, use the **Force** options to generate SWO trace data.

The generated data will be displayed in the Trace window. Choose between:

- | | |
|-----------------------|---|
| Time Stamps | Enables timestamps for various SWO trace packets, that is sent over the SWO communication channel. Use the resolution drop-down list to choose the resolution of the timestamp value. For example, 1 to count every cycle, or 16 to count every 16th cycle. Note that the lowest resolution is only useful if the time between each event packet is long enough. 16 is useful if using a low SWO clock frequency. |
| PC samples | Enables sampling the program counter register, PC, at regular intervals. To choose the sampling rate, see <i>PC Sampling</i> , page 181. |
| Interrupt Logs | Enables generation of interrupt logs. For information about other C-SPY features that also use trace data for interrupts, see <i>Interrupts</i> , page 245. |

Generate

Enables trace data generation for these events. The generated data will be displayed in the Trace window. The value of the counters are displayed in the **Comment** column in the SWO Trace window. Choose between:

CPI	Enables generation of trace data for the CPI counter.
EXC	Enables generation of trace data for the EXC counter.
SLEEP	Enables generation of trace data for the SLEEP counter.
LSU	Enables generation of trace data for the LSU counter.
FOLD	Enables generation of trace data for the FOLD counter.

SWO Configuration

Displays the **SWO Configuration** dialog box where you can configure the hardware’s generation of trace data. See *SWO Configuration dialog box*, page 181.

SWO Configuration dialog box

The **SWO Configuration** dialog box is available from the **J-Link** menu or the **ST-LINK** menu, respectively, alternatively from the **SWO Trace Window Settings** dialog box.

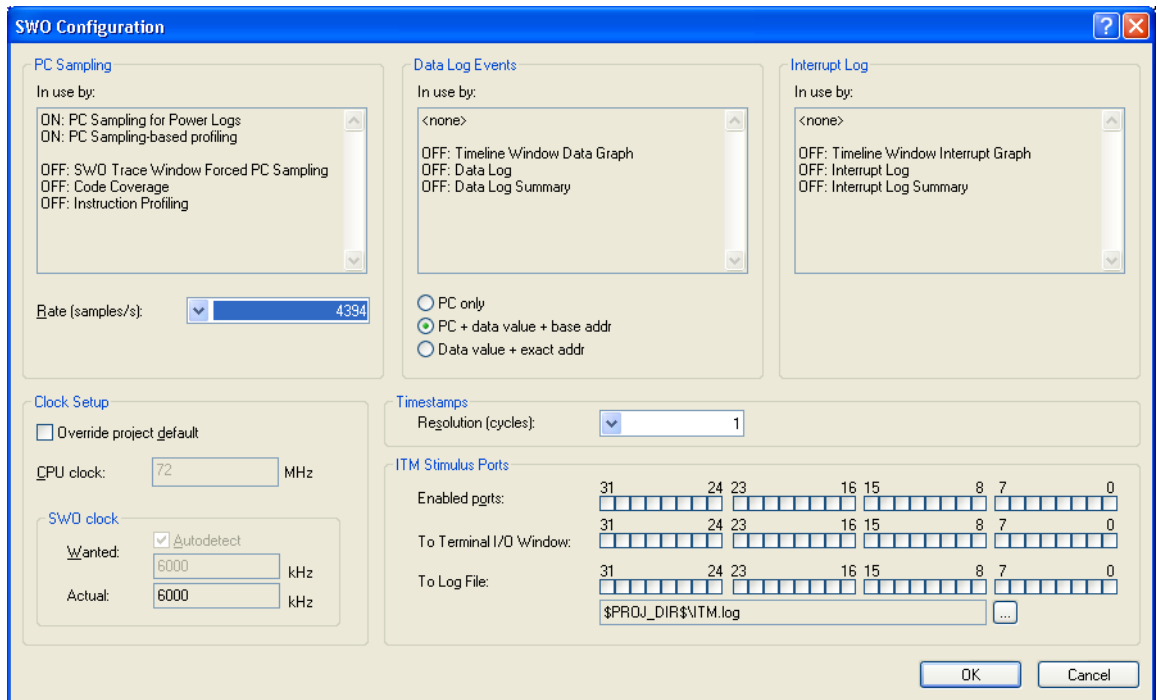


Figure 77: SWO Configuration dialog box

Use this dialog box to configure the serial-wire output communication channel and the hardware's generation of trace data.

See also *Getting started with SWO trace*, page 173.

PC Sampling

Controls the behavior of the sampling of the program counter. You can specify:

In use by

Lists the features in C-SPY that can use trace data for PC Sampling. ON indicates features currently using trace data. OFF indicates features currently not using trace data.

Rate	Use the drop-down list to choose the sampling rate, that is, the number of samples per second. The highest possible sampling rate depends on the SWO clock value and on how much other data that is sent over the SWO communication channel. The higher values in the list will not work if the SWO communication channel is not fast enough to handle that much data.
-------------	--

Data Log Events

Specifies what to log when a Data Log breakpoint is triggered. These items are available:

In use by	Lists the features in C-SPY hat can use trace data for Data Log Events. On indicates features currently using trace data. OFF indicates features currently not using trace data.
PC only	Logs the value of the program counter.
PC + data value + base addr	Logs the value of the program counter, the value of the data object, and its base address.
Data value + exact addr	Logs the value of the data object and the exact address of the data object that was accessed.

Interrupt Log

Lists the features in C-SPY that can use trace data for Interrupt Logs. ON indicates features currently using trace data. OFF indicates features currently not using trace data.

For more information about interrupt logging, see *Interrupts*, page 245.

Override project default

Overrides the **CPU clock** and the **SWO clock** default values on the **Project>Options>J-Link/J-Trace>Setup** page for J-Link/J-Trace or on the **Project>Options>ST-Link>Setup** page for ST-LINK, respectively.

CPU clock

Specify the exact clock frequency used by the internal processor clock, `HCLK`, in MHz. The value can have decimals.

This value is used for configuring the SWO communication speed and for calculating timestamps.

SWO clock

Specify the clock frequency of the SWO communication channel in kHz. Choose between:

Autodetect	Automatically uses the highest possible frequency that the J-Link debug probe can handle. When it is selected, the Wanted text box displays that frequency.
Wanted	Manually selects the frequency to be used, if Autodetect is not selected. The value can have decimals. Use this option if data packets are lost during transmission.
Actual	Displays the frequency that is actually used. This can differ a little from the wanted frequency.

Timestamps

Selects the resolution of the timestamp value. For example, 1 to count every cycle, or 16 to count every 16th cycle. Note that the lowest resolution is only useful if the time between each event packet is long enough.

ITM Stimulus Ports

Selects which ports you want to redirect and to where. The ITM Stimulus Ports are used for sending data from your application to the debugger host without stopping the program execution. There are 32 such ports. Choose between:

Enabled ports	Enables the ports to be used. Only enabled ports will actually send any data over the SWO communication channel to the debugger. Port 0 is used by the terminal I/O library functions. Ports 1-4 are used by the ITM macros for the Event Log window. Port 5 is used for an optional PC value added to the ITM macro.
To Terminal I/O window	Specifies the ports to use for routing data to the Terminal I/O window.
To Log File	Specifies the ports to use for routing data to a log file. To use a different log file than the default one, use the browse button.



The `stdout` and `stderr` of your application can be routed via SWO to the C-SPY Terminal I/O window, instead of via semihosting. To achieve this, choose **Project>Options>General Options>Library Configuration>Library low-level interface implementation>stdout/stderr>Via SWO**. This will significantly improve the performance of `stdout/stderr`, compared to when semihosting is used.

This can be disabled if you deselect the port settings in the **Enabled ports** and **To Terminal I/O** options.

Trace window

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

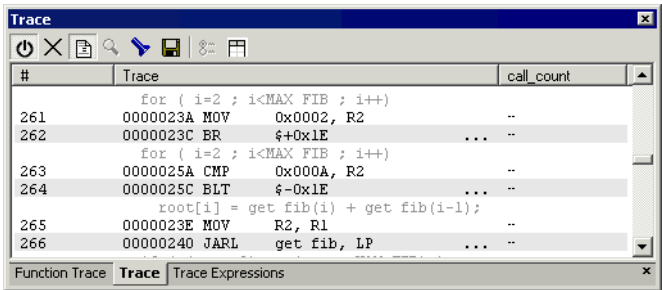


Figure 78: The Trace window in the simulator

Note: There are three different Trace windows—ETM Trace, SWO Trace, and just Trace for the C-SPY simulator. The windows look slightly different.

This window displays the collected trace data, where the content differs depending on the C-SPY driver you are using and the trace support of your debug probe:

- C-SPY simulator

The window displays a collected sequence of executed machine instructions. In addition, the window can display trace data for expressions.
- ETM trace

The window displays the sequence of executed instructions—optionally with embedded source—which has been continuously collected during application execution, that is *full trace*. The data has been collected in the ETM trace buffer. The collected data is displayed after the execution has stopped.

For information about the requirements for using ETM trace, see *Requirements for using ETM trace*, page 172.

SWO trace

The window displays all events transmitted on the SWO channel. The data is streamed from the target system, via the SWO communication channel, and continuously updated live in the Trace window. Note that if you use the SWO communication channel on a trace probe, the data will be collected in the trace buffer and displayed after the execution has stopped.

For information about the requirements for using SWO trace, see *Requirements for using SWO trace*, page 172.

Trace toolbar

The toolbar in the Trace window and in the Function trace window contains:

**Enable/Disable**

Enables and disables collecting and viewing trace data in this window. This button is not available in the Function trace window.

**Clear trace data**

Clears the trace buffer. Both the Trace window and the Function trace window are cleared.

**Toggle source**

Toggles the Trace column between showing only disassembly or disassembly together with the corresponding source code.

**Browse**

Toggles browse mode on or off for a selected item in the Trace window, see *Browsing through trace data*, page 176.

**Find**

Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 209.

**Save**

In the ETM Trace and SWO Trace windows this button displays the Trace Save dialog box, see *Trace Save dialog box*, page 188. In the C-SPY simulator this button 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 ETM Trace window this button displays the **Trace Settings** dialog box, see *ETM Trace Settings dialog box*, page 177.

In the SWO Trace window this buttons displays the **SWO Trace Window Settings** dialog box, see *SWO Trace Window Settings dialog box*, page 179.

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 208.

Display area (in the C-SPY simulator)

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 208.

Display area (for ETM trace)

This area contains these columns for ETM trace:

Index	A number that corresponds to each packet. Examples of packets are instructions, synchronization points, and exception markers.
-------	--

Frame Time	<p>When collecting trace data in cycle-accurate mode (requires ARM7/9)—enable Cycle accurate tracing in the ETM Trace Settings dialog box—the value corresponds to the number of elapsed cycles since the start of the execution. This column is only available for the J-Link/J-Trace driver.</p> <p>When collecting trace data in non-cycle-accurate mode, the value corresponds to an approximate amount of cycles. For Cortex-M devices, the value is repeatedly calibrated with the actual number of cycles.</p> <p>When the Show timestamp option is selected in the ETM Trace Settings dialog box, the value displays the time instead of cycles. To display the value as time requires collecting data in cycle-accurate mode, see <i>Cycle accurate tracing</i>, page 178, and the J-Link/J-Trace driver.</p>
Address	The address of the executed instruction.
Opcode	The operation code of the executed instruction.
Trace	The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.
Comment	This column is only available for the J-Link/J-Trace driver.

Note: For RDI drivers, this window looks slightly different.

Display area (for SWO trace)

This area contains these columns for SWO trace:

Index	An index number for each row in the trace buffer. Simplifies the navigation within the buffer.
SWO Packet	The contents of the captured SWO packet.
Cycles	The approximate number of cycles from the start of the execution until the event.

Event	The event type of the captured SWO packet. If the column displays <i>Overflow</i> , the data packet could not be sent, because too many SWO features use the SWO channel at the same time. To decrease the amount of transmissions on the communication channel, point at the SWO button—on the IDE main window toolbar—with the mouse pointer to get detailed tooltip information about which C-SPY features that have requested trace data generation. Disable some of the features.
Value	The event value, if any.
Trace	If the event is a sampled PC value, the instruction is displayed in this column. Optionally, the corresponding source code can also be displayed.
Comment	Additional information. This includes the values of the selected Trace Events counters, or the number of the comparator (hardware breakpoint) used for the Data Log breakpoint.



If the display area seems to show garbage, make sure you specified a correct value for the **CPU clock** in the **SWO Configuration** dialog box.

Trace Save dialog box

The **Trace Save** dialog box is available from the driver-specific menu, and from the Trace window and the SWO Trace window.

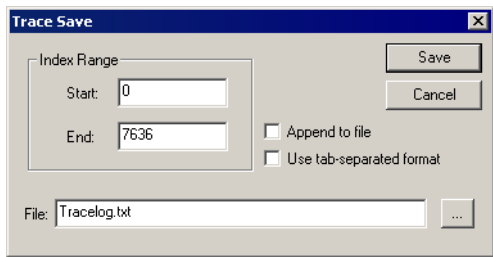


Figure 79: Trace Save dialog box

Index Range

Saves a range of frames to a file. Specify a start index and an end index (as numbered in the index column in the Trace window).

Append to file

Appends the trace data to an existing file.

Use tab-separated format

Saves the content in columns that are tab-separated, instead of separated by white spaces.

File

Specify a file for the trace data.

Function Trace window

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

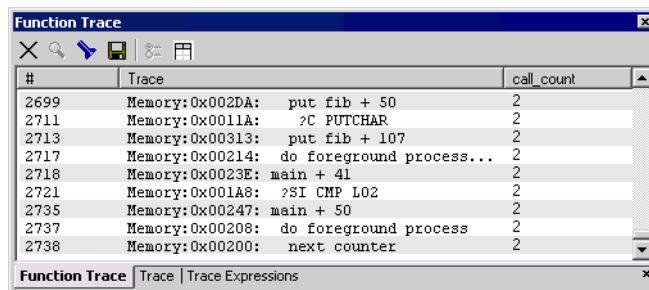


Figure 80: Function Trace window

This window is available for the:

- C-SPY simulator
- J-Trace driver

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 185.

Display area

For information about the columns in the display area, see:

- *Display area (in the C-SPY simulator)*, page 186
- *Display area (for ETM trace)*, page 186.

Timeline window

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

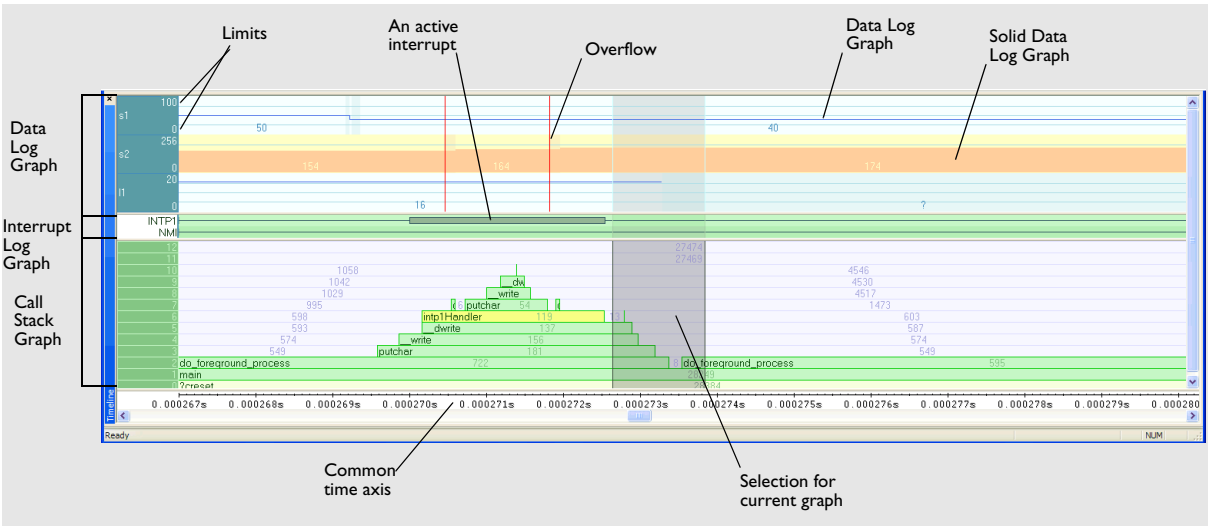


Figure 81: Timeline window

This window is available for the:

- C-SPY simulator
- J-Link/J-Trace driver
- ST-LINK driver.

This window displays trace data (for interrupt logs, data logs, event logs, and for the call stack) as well as logged power values, as graphs in relation to a common time axis.

To display a graph:

- 1 Choose *C-SPYdriver*>**SWO Configuration** to open the **SWO Configuration** dialog box. Make sure the **CPU clock** option is set to the same value as the CPU clock value set by your application. This is necessary to set the SWO clock and to obtain a correct data transfer to the debug probe.

If you are using the C-SPY simulator you can ignore this step.

- 2 Choose **Timeline** from the *C-SPY driver* menu to open the Timeline window.
- 3 In the Timeline window, click in the graph area and choose **Enable** from the context menu to enable a specific graph.
- 4 For the Data Log Graph, you need to set a Data Log breakpoint for each variable you want a graphical representation of in the Timeline window. See *Data Log breakpoints dialog box*, page 130.
- 5 For the Event Log Graph, you must add a preprocessor macro to your application source code where you want event logs to be generated. See *Getting started using event logging*, page 90.
- 6 Click **Go** on the toolbar to start executing your application. The graph appears.

To navigate in the graph, use any of these alternatives:

- Right-click and from the context menu choose **Zoom In** or **Zoom Out**. Alternatively, use the + and - keys. The graph zooms in or out depending on which command you used.
- Right-click in the graph and from the context menu choose **Navigate** and the appropriate command to move backwards and forwards on the graph. Alternatively, use any of the shortcut keys: arrow keys, Home, End, and Ctrl+End.
- Double-click on a sample of interest and the corresponding source code is highlighted in the editor window and in the Disassembly window.
- Click on the graph and drag to select a time interval. Press Enter or right-click and from the context menu choose **Zoom>Zoom to Selection**. The selection zooms in.



Point in the graph with the mouse pointer to get detailed tooltip information for that location.

Display area

Depending on the C-SPY driver you are using, the display area can be populated with different graphs:

Graphs	C-SPY simulator	J-Link driver	J-Trace driver	ST-LINK driver
Interrupt Log Graph	X	X	X	X

Table 13: Supported graphs in the Timeline window

Graphs	C-SPY simulator	J-Link driver	J-Trace driver	ST-LINK driver
Data Log Graph	--	X	X ¹	X
Event Log Graph	--	X	X	X
Call Stack Graph	X	--	X	--
Power Log Graph	--	X	--	--

Table 13: Supported graphs in the Timeline window (Continued)

1 Only available when ETM trace is disabled.

If a specific graph is available or not depends on abilities in hardware, debugger probe, and the C-SPY driver. See Table 3, *Driver differences, J-Link/J-Trace and ST-LINK* on page 38, and *Requirements for using trace*, page 171.

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

Interrupt Log Graph

The Interrupt Log Graph displays interrupts reported by SWO trace or by the C-SPY simulator. In other words, the graph provides a graphical view of the interrupt events during the execution of your application, where:

- The label area at the left end of the graph shows the names of the interrupts.
- The graph itself shows active interrupts as a thick green horizontal bar. This graph is a graphical representation of the information in the Interrupt Log window, see *Interrupt Log window*, page 260.

Data Log Graph

The Data Log Graph displays the data logs generated by SWO trace, for up to four different variables or address ranges specified as Data Log breakpoints, where:

- Each graph is labeled with—in the left-side area—the variable name or address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the Y-axis for a variable. You can use the context menu to change these limits. The graph can be displayed either as a thin line or as a color-filled solid graph. The graph is a graphical representation of the information in the Data Log window, see *Event Log window*, page 104.
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system.

Event Log Graph

The Event Log Graph displays the event logs produced when the execution passes specific positions in your application code, where:

- Each graph is labeled—in the left-side area— with the name of the channel.
- For each channel, there will be a vertical line that indicates when the event occurred. Optionally, you can choose to display the event value that was passed with the event.

Call Stack Graph

The Call Stack Graph displays the sequence of calls and returns collected by ETM 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.

Power Log Graph

The Power Log Graph displays power measurement samples generated by the debug probe or associated hardware.

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:

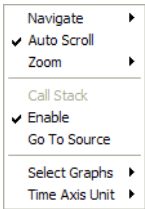


Figure 82: Timeline window context menu for the Call Stack Graph

Note: The context menu contains some commands that are common to all graphs and some commands that are specific to each graph. The figure reflects the context menu for the Call Stack Graph, which means that the menu looks slightly different for the other graphs.

These commands are available:

Navigate	All graphs	Commands for navigating over the graph(s); choose between: Next moves the selection to the next relevant point in the graph. Shortcut key: right arrow. Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow. First moves the selection to the first data entry in the graph. Shortcut key: Home. Last moves the selection to the last data entry in the graph. Shortcut key: End. End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.
----------	------------	---

Auto Scroll	All graphs	Toggles auto scrolling on or off. When on, the most 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>
Data Log	Data Log Graph	A heading that shows that the Data Log-specific commands below are available.
Power Log	Power Log Graph	A heading that shows that the Power Log-specific commands below are available.
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.

Variable	Data Log Graph	The name of the variable for which the Data Log-specific commands below apply. This menu command is context-sensitive, which means it reflects the Data Log Graph you selected in the Timeline window (one of up to four).
Solid Graph	Data Log Graph	Displays the graph as a color-filled solid graph instead of as a thin line.
Viewing Range	Data and Power Log Graph	Displays a dialog box, see <i>Viewing Range dialog box</i> , page 197.
Size	Data and Power Log Graph	Determines the vertical size of the graph; choose between Small , Medium , and Large .
Show Numerical Value	Data and Power Log Graph	Shows the numerical value of the variable, in addition to 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.

Viewing Range dialog box

The **Viewing Range** dialog box is available from the context menu that appears when you right-click in the Power Log Graph or the Data Log Graph in the Timeline window.

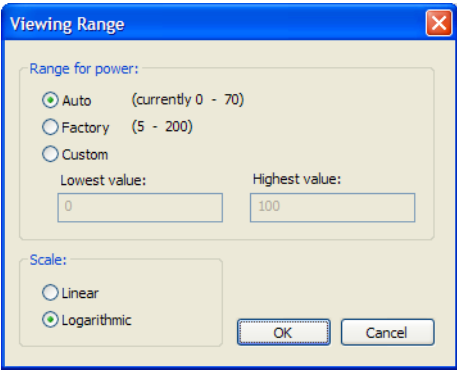


Figure 83: Viewing Range dialog box

Use this dialog box to specify the value range, that is, the range for the Y-axis for the graph.

Range for xxxx

Selects the viewing range for the displayed values:

Auto	Uses the range according to the range of the values that are actually collected, continuously keeping track of minimum or maximum values. The currently computed range, if any, is displayed in parentheses. The range is rounded to reasonably <i>even</i> limits.
Factory	For the Data Log Graph: Uses the range according to the value range of the variable, for example 0–65535 for an unsigned 16-bit integer. For the Power Log Graph: Uses the range according to the properties of the measuring hardware.
Custom	Use the text boxes to specify an explicit range.

Scale

Selects the scale type of the Y-axis:

- **Linear**
- **Logarithmic.**

Trace Start breakpoints dialog box (simulator)

The **Trace Start** dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

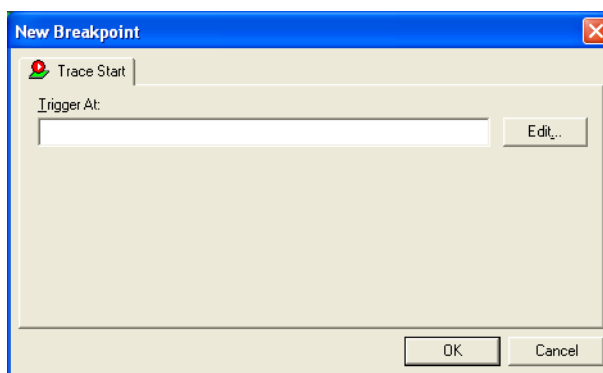


Figure 84: Trace Start breakpoints dialog box

This dialog box is available for the C-SPY simulator. See also *Trace Start breakpoints dialog box*, page 200.

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 135.

Trace Stop breakpoints dialog box (simulator)

The **Trace Stop** dialog box is available from the context menu that appears when you right-click in the Breakpoints window.

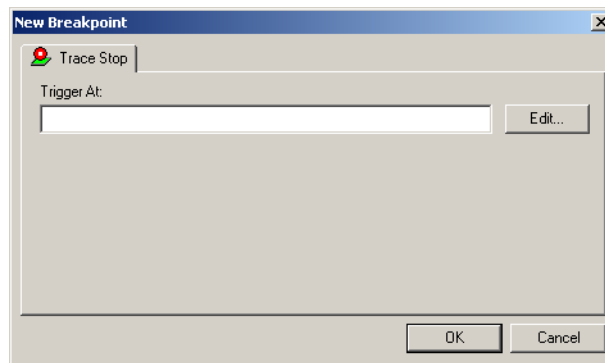


Figure 85: Trace Stop breakpoints dialog box

This dialog box is available for the C-SPY simulator. See also *Trace Stop breakpoints dialog box*, page 203.

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 135.

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. You can also right-click in the editor window or the Disassembly window, and then choose **Toggle Breakpoint (Trace Start)**.

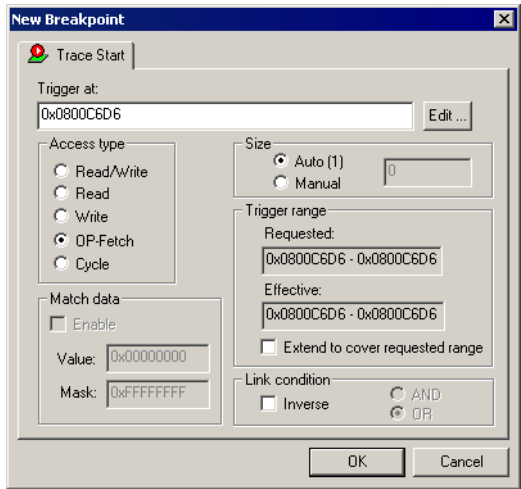


Figure 86: Trace Start breakpoints dialog box (J-Link/J-Trace)

Use this dialog box to set the conditions that determine when to start collecting trace data. When the trace condition is triggered, the trace data collection is started.

This dialog box is available for the C-SPY J-Link/J-Trace driver.

Trigger at

Specify the starting point of the code section for which you want to collect trace data. You can specify a variable name, an address, or a cycle counter value.

Size

Controls the size of the address range, that when reached, will trigger the start of the trace data collection. Choose between:

Auto	Sets the size automatically. This can be useful if Trigger at contains a variable.
Manual	Specify the size of the breakpoint range manually.

Trigger range

Shows the requested range and the effective range to be covered by the trace data collection. The range suggested is either within or exactly the area specified by the **Trigger at** and the **Size** options.

Extend to cover requested range	<p>Extends the range so that a whole data structure is covered. For data structures that do not fit the size of the possible ranges supplied by the hardware breakpoint unit, for example three bytes, the range will not cover the whole data structure. Note that the range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data.</p> <p>This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.</p>
--	--

Access type

Specifies the type of memory access that triggers the trace data collection. Choose between:

Read/Write	Read from or write to location.
Read	Read from location.
Write	Write to location.
OP-fetch	At execution address
Cycle	The number of counter cycles at a specific point in time, counted from where the execution started. This option is only available for Cortex-M devices.

Match data

Enables matching of the accessed data. Use the **Match data** options in combination with the Read/Write, Read, or Write access types for data. This option can be useful when you want a trigger when a variable has a certain value.

Value	Specify a data value.
Mask	Specify which part of the value to match (word, halfword, or byte).

The **Match data** options are only available for J-Link/J-Trace and when using a Cortex-M device.

Note: For Cortex-M devices, only one breakpoint with **Match data** can be set. Such a breakpoint uses two breakpoint resources.

Link condition

Specifies how trace conditions are combined, using **AND** and **OR**. When combining a condition that has the link condition **AND** with a condition that has the link condition **OR**, **AND** has precedence. The option **Inverse** inverts the trace condition and is individual for each trace filter condition. If one trace start or stop condition is inverted, all others will be too. An inverted trace start or stop condition means that the trace data collection is performed everywhere except for this section of the application code.

For ARM7/9 devices, trace filters are combined using the OR algorithm. Use the **Inverse** option to invert the trace filter; all trace filters are affected. The trace filter will be combined with the start and stop triggers, if any, using the AND algorithm.

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. You can also right-click in the editor window or the Disassembly window, and then choose **Toggle Breakpoint (Trace Stop)**.

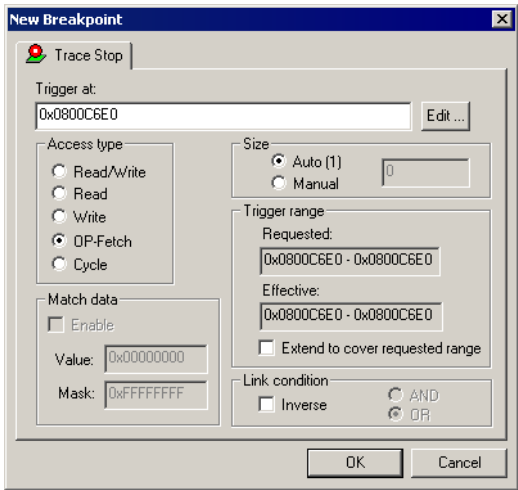


Figure 87: Trace Stop breakpoints dialog box (J-Link/J-Trace)

When the trace condition is triggered, the trace data collection is performed for some further instructions, and then the collection is stopped.

This dialog box is available for the C-SPY J-Link/J-Trace driver.

Trigger at

Specify the stopping point of the code section for which you want to collect trace data. You can specify a variable name, an address, or a cycle counter value.

Size

Controls the size of the address range, that when reached, will trigger the stop of the trace data collection. Choose between:

- | | |
|---------------|---|
| Auto | Sets the size automatically. This can be useful if Trigger at contains a variable. |
| Manual | Specify the size of the breakpoint range manually. |

Trigger range

Shows the requested range and the effective range to be covered by the trace data collection. The range suggested is either within or exactly the area specified by the **Trigger at** and the **Size** options.

Extend to cover requested range	Extends the range so that a whole data structure is covered. For data structures that do not fit the size of the possible ranges supplied by the hardware breakpoint unit, for example three bytes, the range will not cover the whole data structure. Note that the range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data. This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.
--	---

Access type

Specifies the type of memory access that triggers the trace data collection. Choose between:

Read/Write	Read from or write to location.
Read	Read from location.
Write	Write to location.
OP-fetch	At execution address
Cycle	The number of counter cycles at a specific point in time, counted from where the execution started. This option is only available for Cortex-M devices.

Match data

Enables matching of the accessed data. Use the **Match data** options in combination with the Read/Write, Read, or Write access types for data. This option can be useful when you want a trigger when a variable has a certain value.

Value	Specify a data value.
Mask	Specify which part of the value to match (word, halfword, or byte).

The **Match data** options are only available for J-Link/J-Trace and when using a Cortex-M device.

Note: For Cortex-M devices, only one breakpoint with **Match data** can be set. Such a breakpoint uses two breakpoint resources.

Link condition

Specifies how trace conditions are combined, using **AND** and **OR**. When combining a condition that has the link condition **AND** with a condition that has the link condition **OR**, **AND** has precedence. The option **Inverse** inverts the trace condition and is individual for each trace filter condition. If one trace start or stop condition is inverted, all others will be too. An inverted trace start or stop condition means that the trace data collection is performed everywhere except for this section of the application code.

For ARM7/9 devices, trace filters are combined using the OR algorithm. Use the **Inverse** option to invert the trace filter; all trace filters are affected. The trace filter will be combined with the start and stop triggers, if any, using the AND algorithm.

Trace Filter breakpoints dialog box

The **Trace Filter** dialog box is available from the context menu that appears when you right-click in the Breakpoints window. You can also right-click in the editor window or the Disassembly window, and then choose **Toggle Breakpoint (Trace Filter)**.

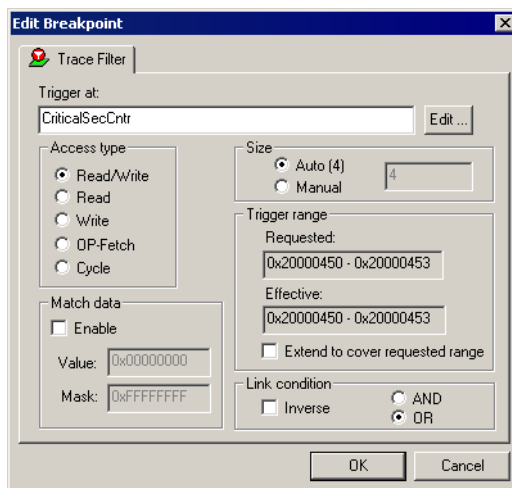


Figure 88: Trace Filter breakpoints dialog box

This dialog box is available for the J-Trace driver.

When the trace condition is triggered, the trace data collection is performed for some further instructions, and then the collection is stopped.

Trigger at

Specify the code section for which you want to collect trace data. You can specify a variable name, an address, or a cycle counter value.

Size

Controls the size of the address range where filtered trace is active. Choose between:

- | | |
|---------------|---|
| Auto | Sets the size automatically. This can be useful if Trigger at contains a variable. |
| Manual | Specify the size of the range manually. |

Trigger range

Shows the requested range and the effective range to be covered by the filtered trace data collection. The range suggested is either within or exactly the area specified by the **Trigger at** and the **Size** options.

- | | |
|--|--|
| Extend to cover requested range | <p>Extends the range so that a whole data structure is covered. For data structures that do not fit the size of the possible ranges supplied by the hardware breakpoint unit, for example three bytes, the range will not cover the whole data structure. Note that the range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data.</p> <p>This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.</p> |
|--|--|

Access type

Specifies the type of memory access that activates the trace data collection. Choose between:

- | | |
|-------------------|---------------------------------|
| Read/Write | Read from or write to location. |
| Read | Read from location. |
| Write | Write to location. |

OP-fetch	At execution address
Cycle	The number of counter cycles at a specific point in time, counted from where the execution started. This option is only available for Cortex-M devices.

Match data

Enables matching of the accessed data. Use the **Match data** options in combination with the Read/Write, Read, or Write access types for data. This option can be useful when you want a trigger when a variable has a certain value.

Value	Specify a data value.
Mask	Specify which part of the value to match (word, halfword, or byte).

The **Match data** options are only available for J-Link/J-Trace and when using a Cortex-M device.

Note: For Cortex-M devices, only one breakpoint with **Match data** can be set. Such a breakpoint uses two breakpoint resources.

Link condition

Specifies how trace conditions are combined, using **AND** and **OR**. When combining a condition that has the link condition **AND** with a condition that has the link condition **OR**, **AND** has precedence. The option **Inverse** inverts the trace condition and is individual for each trace filter condition. If one trace start or stop condition is inverted, all others will be too. An inverted trace start or stop condition means that the trace data collection is performed everywhere except for this section of the application code.

For ARM7/9 devices, trace filters are combined using the OR algorithm. Use the **Inverse** option to invert the trace filter; all trace filters are affected. The trace filter will be combined with the start and stop triggers, if any, using the AND algorithm.

Trace Expressions window

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

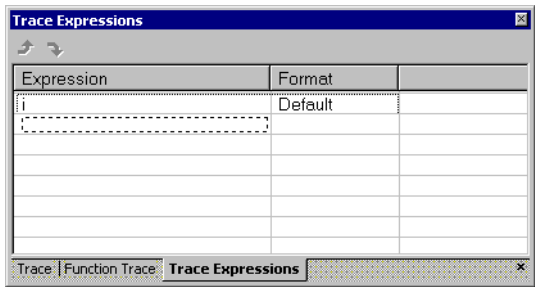


Figure 89: Trace Expressions window

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.

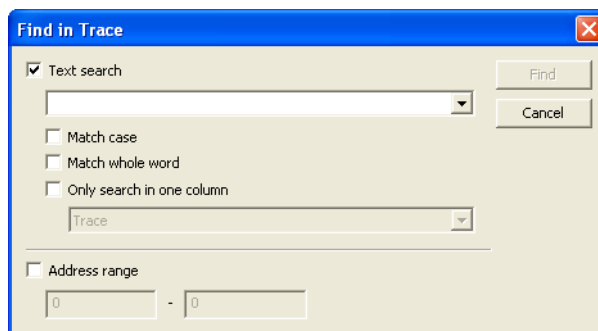


Figure 90: Find in Trace dialog box

This dialog box is available for the:

- C-SPY simulator
- J-Link/J-Trace driver
- ST-LINK driver, when SWO is enabled.

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 210.

See also *Searching in trace data*, page 175.

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 <code>int</code> will also find <code>INT</code> and <code>Int</code> and so on. |
| Match whole word | Searches only for the string when it occurs as a separate word. Otherwise <code>int</code> will also find <code>print</code> , <code>sprintf</code> and so on. |

Only search in one column Searches only in the column you selected from the drop-down list.

Address Range

Specify the address range you want to display or search. The trace data within the address range is displayed. If you also have specified a text string in the **Text search** field, the text string is searched for within the address range.

Find in Trace window

The Find in Trace window is available from the **View>Messages** menu. Alternatively, it is automatically displayed when you perform a search using the **Find in Trace** dialog box or perform a search using the **Find in Trace** command available from the context menu in the editor window.

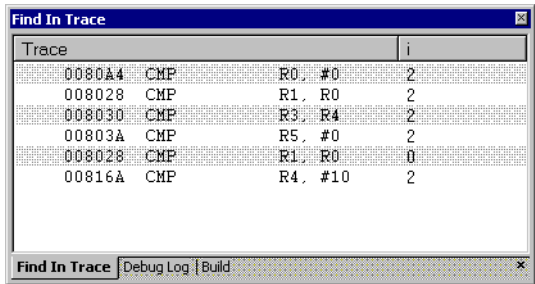


Figure 91: Find in Trace window

This dialog box is available for the:

- C-SPY simulator
- J-Link/J-Trace driver
- ST-LINK driver, when SWO is enabled.

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 209.

For more information, see *Searching in trace data*, page 175.

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.

To profile only a specific part of your code, you can select a *time interval*—using the Timeline window—for which C-SPY produces profiling information. Alternatively, you can use *filtered profiling*, which means that you can exclude, for example, individual functions from being profiled.

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 (ETM trace) is analyzed to determine all function calls and returns. When the collected instruction sequence is incomplete or discontinuous, as sometimes happens when using ETM trace, the profiling information is less accurate.

- Trace (flat) / Sampling

Each instruction in the full instruction trace (ETM trace) or each PC Sample (from SWO trace) 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.

- Breakpoints

The profiler sets a breakpoint on every function entry point. During execution, the profiler collects information about function calls and returns as each breakpoint is hit. This assumes that the hardware supports a large number of breakpoints, and it has a huge impact on execution performance.

Power sampling

Some debug probes support regular sampling of the power consumption of the development board, or components on the board. Each sample is also associated with a PC sample and represents the power consumption (actually, the electrical current) for a small time interval preceding the time of the sample. When the profiler is set to use *Power Sampling*, additional columns are displayed in the Profiler window. Each power sample is associated with a function or code fragment, just as with regular PC Sampling. Note that this does not imply that all the energy corresponding to a sample can be attributed to that function or code fragment. The time scales of power samples and instruction execution are vastly different; during one power measurement, the CPU has typically executed many thousands of instructions. Power Sampling is a statistics tool.

REQUIREMENTS FOR USING THE PROFILER

The C-SPY simulator supports the profiler, and there are no specific requirements for using the profiler.

To use the profiler in your hardware debugger system, you need one of these setups:

- A J-Link, a J-Trace, ST-LINK debug probe with an SWD/SWO interface between the probe and the target system, which must be based on a Cortex-M device
- A J-Trace debug probe and an ARM7/9 device with ETM trace.
- A J-Link or J-Trace Ultra probe.

This table lists the C-SPY driver profiling support:

C-SPY driver	Trace (calls)	Trace (flat)	Sampling	Power
C-SPY simulator	X	X	--	--
J-Link	--	--	X*	--
J-Link Ultra	--	--	X*	X
J-Trace	X	X	X*	--
JTAGjet	--	--	--	--
RDI	--	--	--	--
Macraigor	--	--	--	--
GDB Server	--	--	--	--
ST-LINK	--	--	X*	--
TI Stellaris	--	--	--	--
TI XDS100	--	--	--	--
Angel	--	--	--	--
IAR ROM-monitor	--	--	--	--

Table 14: C-SPY driver profiling support

* Only for Cortex-M devices.

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
- Selecting a time interval for profiling information.


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>Include debug information in output

Table 15: Project options for enabling the profiler

- 2 To set up the profiler for function profiling:
 - If you use ETM trace, make sure that the **Cycle accurate tracing** option is selected in the **Trace Settings** dialog box.
 - If you use the SWD/SWO interface, no specific settings are required.
- 3 When you have built your application and started C-SPY, choose **J-Link>Function Profiler** to open the Function Profiler window, and click the **Enable** button to turn on the profiler. Alternatively, choose **Enable** from the context menu that is available when you right-click in the Function Profiler window.
- 4 Start executing your application to collect the profiling information.
- 5 Profiling information is displayed in the Function Profiler window. To sort, click on the relevant column header.
-  6 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.

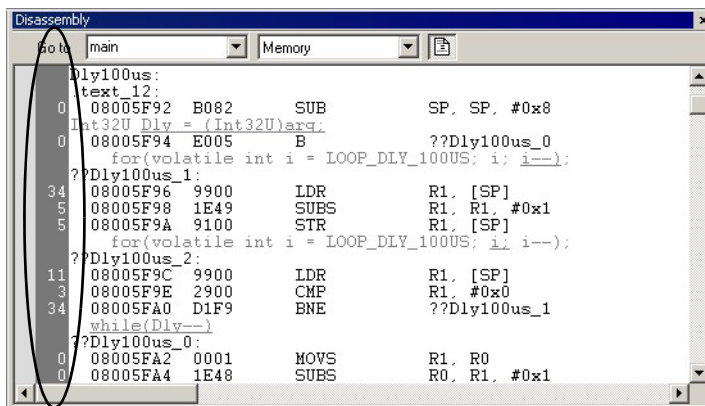


Figure 92: Instruction count in Disassembly window

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.

SELECTING A TIME INTERVAL FOR PROFILING INFORMATION

Normally, the profiler computes its information from all PC samples it receives, accumulating more and more information until you explicitly clear the profiling information. However, you can choose a time interval for which the profiler computes the PC samples. This function is supported by the J-Link probe, the J-Trace probe and the ST-LINK probe.

To select a time interval, follow these steps:

- 1 Choose **Function Profiler** from the **J-Link** menu.
- 2 In the Function Profiler window, right-click and choose **Source: Sampling** from the context menu.
- 3 Execute your application to collect samples.
- 4 Choose **View>Timeline**.

5 In the Timeline window, click and drag to select a time interval.

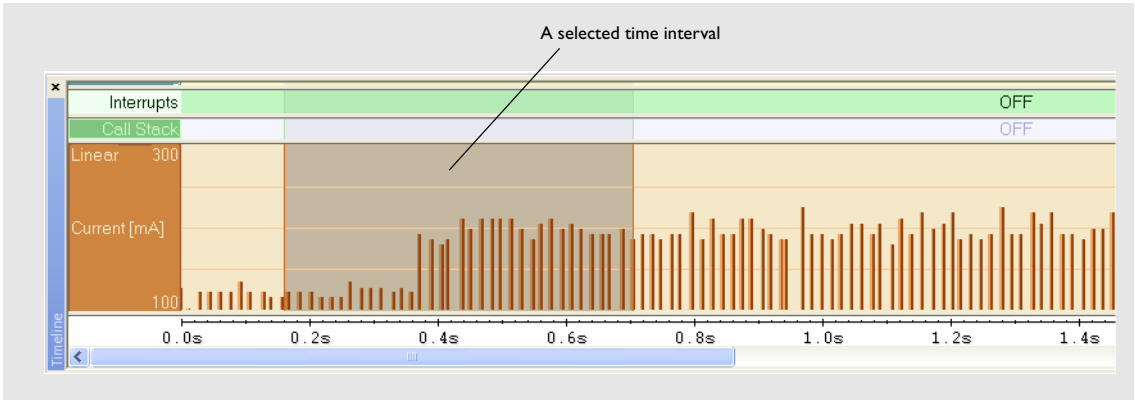


Figure 93: Power Graph with a selected time interval

6 In the selected time interval, right-click and choose **Profile Selection** from the context menu.

The Function Profiler window now displays profiling information for the selected time interval.

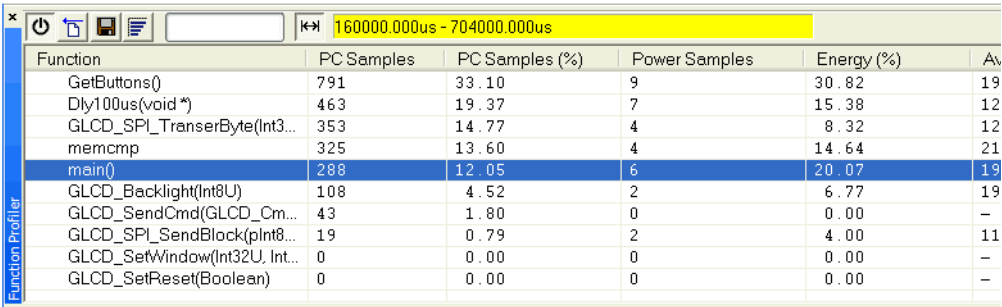


Figure 94: Function Profiler window in time-interval mode

7 Click the **Full/Time-interval profiling** button to toggle the Full profiling view.

Reference information on the profiler

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

- *Function Profiler window*, page 219

- *Disassembly window*, page 71

See also:

- *ETM Trace Settings dialog box*, page 177
- *SWO Trace Window Settings dialog box*, page 179
- *SWO Configuration dialog box*, page 181.

Function Profiler window

The Function Profiler window is available from the **J-Link** menu or the **ST-LINK** menu, respectively.

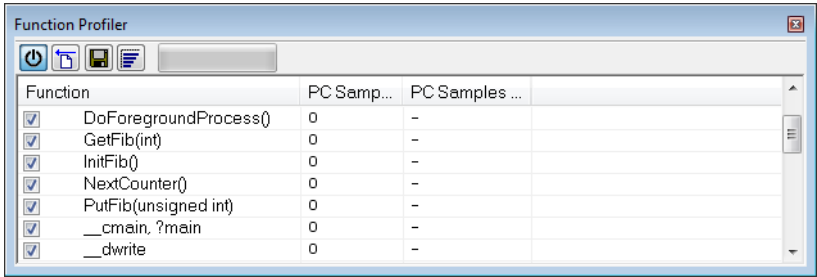


Figure 95: Function Profiler window

This window is available in the:

- C-SPY simulator
- J-Link/J-Trace driver
- ST-LINK driver.

This window displays function profiling information.

When Trace(flat) or Sampling is selected, a checkbox appears on each line in the left-side margin of the window. Use these checkboxes to include or exclude lines from the profiling. Excluded lines are dimmed but not removed.

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.



Time-interval mode

Toggles between profiling a selected time interval or full profiling. This toolbar button is only available if PC Sampling is supported by the debug probe.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 215.

Status field

Displays the range of the selected time interval, in other words, the profiled selection. This field is yellow when Time-interval profiling mode is enabled. This field is only available if PC Sampling is supported by the debug probe (SWO trace).

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 215.

Display area

The content in the display area depends on which source that is used for the profiling information:

- For the Breakpoints and Trace (calls) sources, 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 Sampling and Trace (flat) sources, 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 215.

More specifically, the display area provides information in these columns:

Function	All sources	The name of the profiled C function. For Sampling source, also sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels, is displayed.
Calls	Breakpoints and Trace (calls)	The number of times the function has been called.
Flat time	Breakpoints and Trace (calls)	The time in cycles spent inside the function.
Flat time (%)	Breakpoints and Trace (calls)	Flat time expressed as a percentage of the total time.
Acc. time	Breakpoints and Trace (calls)	The time in cycles spent in this function and everything called by this function.
Acc. time (%)	Breakpoints and Trace (calls)	Accumulated time expressed as a percentage of the total time.
PC Samples	Trace (flat) and Sampling	The number of PC samples associated with the function.
PC Samples (%)	Trace (flat) and Sampling	The number of PC samples associated with the function as a percentage of the total number of samples.
Power Samples	Power Sampling	The number of power samples associated with that function.

Energy (%)	Power Sampling	The accumulated value of all measurements associated with that function, expressed as a percentage of all measurements.
Avg Current [mA]	Power Sampling	The average measured value for all samples associated with that function.
Min Current [mA]	Power Sampling	The minimum measured value for all samples associated with that function.
Max Current [mA]	Power Sampling	The maximum measured value for all samples associated with that function.

Context menu

This context menu is available:

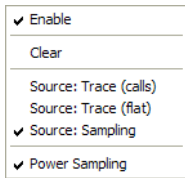


Figure 96: Function Profiler window context menu

These commands are available:

Enable	Enables the profiler. The system will collect information also when the window is closed.
Clear	Clears all profiling data.
Filtering	Selects which part of your code to profile. Choose between: Check All —Excludes all lines from the profiling. Uncheck All —Includes all lines in the profiling. Load —Reads all excluded lines from a saved file. Save —Saves all excluded lines to a file. Typically, this can be useful if you are a group of engineers and want to share sets of exclusions.

These commands are only available when using one of the modes Trace(flat) or Sampling.

Source *	<p>Selects which source to be used for the profiling information. Choose between:</p> <p>Sampling—the instruction count for instruction profiling represents the number of samples for each instruction.</p> <p>Trace (calls)—the instruction count for instruction profiling is only as complete as the collected trace data.</p> <p>Trace (flat)—the instruction count for instruction profiling is only as complete as the collected trace data.</p>
Power Sampling	<p>Toggles power sampling information on or off. This command is supported by the J-Link and J-Trace Ultra debug probes.</p>

* 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 215.

Debugging in the power domain

This chapter describes techniques for power debugging and how you can use C-SPY® to find source code constructions that result in unexpected power consumption. More specifically, this means:

- Introduction to power debugging
- Optimizing your source code for power consumption
- Procedures for power debugging
- Reference information on power debugging.

Introduction to power debugging

This section covers these topics:

- Reasons for using power debugging
- Briefly about power debugging
- Requirements for power debugging.

REASONS FOR USING POWER DEBUGGING

Long battery lifetime is a very important factor for many embedded systems in almost any market segment: medical, consumer electronics, home automation, etc. The power consumption in these systems does not only depend on the hardware design, but also on how the hardware is used. The system software controls how it is used.

For examples of when power debugging can be useful, see *Optimizing your source code for power consumption*, page 227.

BRIEFLY ABOUT POWER DEBUGGING

Power debugging is based on the ability to sample the power consumption—more precisely, the power being consumed by the CPU and the peripheral units—and correlate each sample with the application's instruction sequence and hence with the source code and various events in the program execution.

Traditionally, the main software design goal has been to use as little memory as possible. However, by correlating your application's power consumption with its source code you can get insight into how the software affects the power consumption, and thus how it can be minimized.

Measuring power consumption

The power consumption is measured by the debug probe: The J-Link/J-Trace Ultra debug probe measures the voltage drop across a small resistor in series with the supply power to the device. The voltage drop is measured by a differential amplifier and then sampled by an AD converter.

- You can specify a threshold and an appropriate action to be executed when the threshold value is reached. This means that you can enable or disable the power measurement or you can stop the application's execution and determine the cause of unexpected power values.

Power debugging using C-SPY

C-SPY provides an interface for configuring your power debugging and a set of windows for viewing the power values:

- The Power Setup window is where you can specify a threshold and an action to be executed when the threshold is reached.
- The Power Log window displays all logged power values. This window can be used for finding peaks in the power logging and because the values are correlated with the executed code, you can double-click on a value in the Power Log window to get the corresponding code. The precision depends on the frequency of the samples, but there is a good chance that you find the source code sequence that caused the peak.
- The Timeline window displays power values on a time scale. This provides a convenient way of viewing the power consumption in relation to the other information displayed in the window. The Timeline window is correlated to both the Power Log window, the source code window, and the Disassembly window, which means you are just a double-click away from the source code that corresponds to the values you see on the timeline.
- The Function Profiler window combines the function profiling with the power logging to display the power consumption per function—power profiling. You will get a list of values per function and also the average values together with max and min values. Thus, you will find the regions in the application that you should focus when optimizing for power consumption.

REQUIREMENTS FOR POWER DEBUGGING

To use the features in C-SPY for power debugging, you need:

- A Cortex-M3 device with SWO
- A J-Link debug probe or a J-Link Ultra debug probe. Note that the J-Link probe has very limited accuracy and a low resolution.

Optimizing your source code for power consumption

This section gives some examples where power debugging can be useful and thus hopefully help you identify source code constructions that can be optimized for low power consumption.

WAITING FOR DEVICE STATUS

One common construction that could cause unnecessary power consumption is to use a poll loop for waiting for a status change of, for example a peripheral device.

Constructions like this example execute without interruption until the status value changes into the expected state.

```
while (USBD_GetState() < USBD_STATE_CONFIGURED);
while ((BASE_PMC->PMC_SR & MC_MCKRDY) != PMC_MCKRDY);
```

To minimize power consumption, rewrite polling of a device status change to use interrupts if possible, or a timer interrupt so that the CPU can sleep between the polls.

SOFTWARE DELAYS

A software delay might be implemented as a `for` or `while` loop like for example:

```
i = 10000; /* A software delay */
do i--;
while (i != 0);
```

Such software delays will keep the CPU busy with executing instructions performing nothing except to make the time go by. Time delays are much better implemented using a hardware timer. The timer interrupt is set up and after that, the CPU goes down into a low power mode until it is awakened by the interrupt.

DMA VERSUS POLLED I/O

DMA has traditionally been used for increasing transfer speed. For MCUs there are plenty of DMA techniques to increase flexibility, speed, and to lower power consumption. Sometimes, CPUs can even be put into sleep mode during the DMA transfer. Power debugging lets you experiment and see directly in the debugger what effects these DMA techniques will have on power consumption compared to a traditional CPU-driven polled solution.

LOW-POWER MODE DIAGNOSTICS

Many embedded applications spend most of their time waiting for something to happen: receiving data on a serial port, watching an I/O pin change state, or waiting for a time delay to expire. If the processor is still running at full speed when it is idle, battery life is consumed while very little is being accomplished. So in many applications, the microprocessor is only active during a very small amount of the total time, and by placing it in a low-power mode during the idle time, the battery life can be extended considerably.

A good approach is to have a task-oriented design and to use an RTOS. In a task-oriented design, a task can be defined with the lowest priority, and it will only execute when there is no other task that needs to be executed. This idle task is the perfect place to implement power management. In practice, every time the idle task is activated, it sets the microprocessor into a low-power mode. Many microprocessors and other silicon devices have a number of different low-power modes, in which different parts of the microprocessor can be turned off when they are not needed. The oscillator can for example either be turned off or switched to a lower frequency. In addition, individual peripheral units, timers, and the CPU can be stopped. The different low-power modes have different power consumption based on which peripherals are left turned on. A power debugging tool can be very useful when experimenting with different low-level modes.

You can use the Function profiler in C-SPY to compare power measurements for the task or function that sets the system in a low-power mode when different low-power modes are used. Both the mean value and the percentage of the total power consumption can be useful in the comparison.

CPU FREQUENCY

Power consumption in a CMOS MCU is theoretically given by the formula:

$$P = f * U^2 * k$$

where f is the clock frequency, U is the supply voltage, and k is a constant.

Power debugging lets you verify the power consumption as a factor of the clock frequency. A system that spends very little time in sleep mode at 50 MHz is expected to spend 50% of the time in sleep mode when running at 100 MHz. You can use the power data collected in C-SPY to verify the expected behavior and if there is unlinear dependency on the clock frequency, make sure to choose the operating frequency that gives the lowest power consumption.

DETECTING MISTAKENLY UNATTENDED PERIPHERALS

Peripheral units can consume much power even when they are not actively in use. If you are designing for low power, it is important that you disable the peripheral units and not just leave them unattended when they are not in use. But for different reasons, a peripheral unit can be left with its power supply on; it can be a careful and correct design decision, or it can be an inadequate design or just a mistake. If not the first case, then more power than expected will be consumed by your application. This will be easily revealed by the Power graph in the Timeline window. Double-clicking in the Timeline window where the power consumption is unexpectedly high will take you to the corresponding source code and disassembly code. In many cases, it is enough to disable the peripheral unit when it is inactive, for example by turning off its clock which in most cases will shut down its power consumption completely.

However, there are some cases where clock gating will not be enough. Analog peripherals like converters or comparators can consume a substantial amount of power even when the clock is turned off. The Timeline window will reveal that turning off the clock was not enough and that you need to turn off the peripheral completely.

PERIPHERAL UNITS IN AN EVENT-DRIVEN SYSTEM

Consider a system where one task uses an analog comparator while executing, but the task is suspended by a higher-priority task. Ideally, the comparator should be turned off when the task is suspended and then turned on again once the task is resumed. This would minimize the power being consumed during the execution of the high-priority task.

This is a schematic diagram of the power consumption of an assumed event-driven system where the system at the point of time t_0 is in an inactive mode and the current is I_0 :



Figure 97: Power consumption in an event-driven system

At t_1 , the system is activated whereby the current rises to I_1 which is the system's power consumption in active mode when at least one peripheral device turned on, causing the current to rise to I_1 . At t_2 , the execution becomes suspended by an interrupt which is handled with high priority. Peripheral devices that were already active are not turned off, although the task with higher priority is not using them. Instead, more peripheral devices are activated by the new task, resulting in an increased current I_2 between t_2 and t_3 where control is handed back to the task with lower priority.

The functionality of the system could be excellent and it can be optimized in terms of speed and code size. But also in the power domain, more optimizations can be made. The shadowed area represents the energy that could have been saved if the peripheral devices that are not used between t_2 and t_3 had been turned off, or if the priorities of the two tasks had been changed.

If you use the Timeline window, you can make a closer examination and identify that unused peripheral devices were activated and consumed power for a longer period than necessary. Naturally, you must consider whether it is worth it to spend extra clock cycles to turn peripheral devices on and off in a situation like in the example.

FINDING CONFLICTING HARDWARE SETUPS

To avoid floating inputs, it is a common design practice to connect unused MCU I/O pins to ground. If your source code by mistake configures one of the grounded I/O pins as a logical 1 output, a high current might be drained on that pin. This high unexpected current is easily observed by reading the current value from the Power Graph in the Timeline window. It is also possible to find the corresponding erratic initialization code by looking at the Power Graph at application startup.

A similar situation arises if an I/O pin is designed to be an input and is driven by an external circuit, but your code incorrectly configures the input pin as output.

ANALOG INTERFERENCE

When mixing analog and digital circuits on the same board, the board layout and routing can affect the analog noise levels. To ensure accurate sampling of low-level analog signals, it is important to keep noise levels low. Obtaining a well-mixed signal design requires careful hardware considerations. Your software design can also affect the quality of the analog measurements. Performing a lot of I/O activity at the same time as sampling analog signals causes many digital lines to toggle state at the same time, which might introduce extra noise into the AD converter.

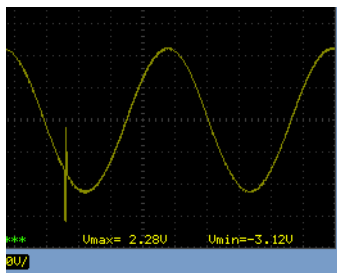


Figure 98: A noise spike recorded by an oscilloscope

Power debugging will help you investigate interference from digital and power supply lines into the analog parts. Power spikes in the vicinity of AD conversions could be the source of noise and should be investigated. All data presented in the Timeline window is correlated to the executed code. Simply double-clicking on a suspicious power value will bring up the corresponding C source code.

Procedures for power debugging

This section gives you step-by-step descriptions of how to use features related to power debugging.

More specifically, you will get information about:

- Displaying the application's power profile and analyzing the result
- Detecting unexpected power usage during application execution.

See also:

- *Timeline window*, page 190
- *Selecting a time interval for profiling information*, page 217.

DISPLAYING THE APPLICATION'S POWER PROFILE AND ANALYZING THE RESULT

To view the power profile:

- 1** Choose **driver-menu>SWO Configuration** to open the **SWO Configuration** dialog box. Make sure the **CPU clock** option is set to the same value as the CPU clock value set by your application. This is necessary to set the SWO clock and to obtain a correct data transfer to the debug probe.
- 2** Choose **C-SPY driver>Timeline** to open the Timeline window.
- 3** Right-click in the graph area and choose **Enable** from the context menu to enable the Power graph.
- 4** Choose **C-SPY driver>Power Log** to open the Power Log window.
- 5** Optionally, if you want to correlate power values to specific interrupts or variables, right-click in the Interrupts or Data Logs graph area, respectively, and choose **Enable** from the context menu.

For variables, you also need to set a Data Log breakpoint for each variable you want a graphical representation of in the Timeline window. See *Data Log breakpoints dialog box*, page 130.

- 6** Optionally, before you start executing your application you can configure the viewing range of the Y-axis for the graph. See *Viewing Range dialog box*, page 197.
- 7** Click **Go** on the toolbar to start executing your application. In the Power Log window, all power values are displayed. In the Timeline window, you will see a graphical representation of the power values, and of the data and interrupt logs if you enabled these graphs. For information about how to navigate on the graph, see *Timeline window*, page 190.

8 To analyze power consumption:

- Double-click on an interesting power value to highlight the corresponding source code is highlighted in the editor window and in the Disassembly window. The corresponding log is highlighted in the Power Log window. For examples of when this can be useful, see *Optimizing your source code for power consumption*, page 227.
- You can identify peripheral units that can be disabled when not used. You can detect this by analyzing the Power graph in combination with the other graphs in the Timeline window. See also *Detecting mistakenly unattended peripherals*, page 229.
- For a specific interrupt, you can see whether the power consumption is changed in an unexpected way after the interrupt exits, for example, if the interrupt enables a power-intensive unit and does not turn it off before exit.
- For function profiling, see *Selecting a time interval for profiling information*, page 217.

DETECTING UNEXPECTED POWER USAGE DURING APPLICATION EXECUTION

To detect unexpected power consumption:

- 1 Choose *C-SPY driver*>**SWO Configuration** to open the **SWO Configuration** dialog box. Make sure the **CPU clock** option is set to the same value as the CPU clock value set by your application. This is necessary to set the SWO clock and to obtain a correct data transfer to the debug probe.
- 2 Choose *C-SPY driver*>**Power Setup** to open the Power Setup window.
- 3 In the Power Setup window, specify a threshold value and the appropriate action, for example **Log All and Halt CPU Above Threshold**.
- 4 Choose *C-SPY driver*>**Power Log** to open the Power Log window. If you continuously want to save the power values to a file, choose **Choose Live Log File** from the context menu. In this case you also need to choose **Enable Live Logging to**.
- 5 Start the execution.

When the power consumption passes the threshold value, the execution will stop and perform the action you specified.

If you saved your logged power values to a file, you can open that file in an external tool for further analysis.

Reference information on power debugging

This section gives reference information for windows and dialog boxes related to power debugging:

- *Power Setup window*, page 234
- *Power Log window*, page 236.

See also:

- *Trace window*, page 184
- *Timeline window*, page 190
- *Viewing Range dialog box*, page 197
- *Function Profiler window*, page 219.

Power Setup window

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

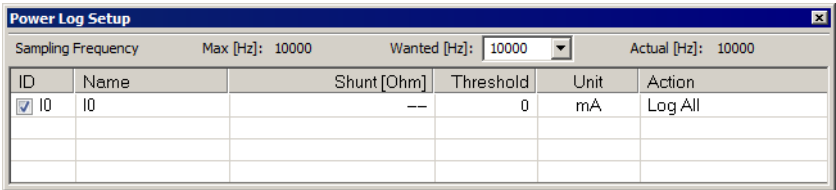


Figure 99: Power Setup window

This window is available for the J-Link driver.

Use this window to configure the power measurement.

Note: To enable power logging, choose **Enable** from the context menu in the Power Log window or from the context menu in the Power Log Graph in the Timeline window.

Display area

This area contains these columns:

- ID
- A unique string that identifies the measurement channel in the probe. Use the check box to activate the channel. If the check box is deselected, logs will not be generated for that channel.

Name	Specify a user-defined name.
Shunt [Ohm]	This column always contains --.
Threshold	Specify a threshold value in the selected unit. The action you specify will be executed when the threshold value is reached.
Unit	Selects the unit for power display. Choose between: nA , uA , or mA .
Action	Displays which action has been selected for the measurement channel. Choose between: Log All , Log Above Threshold , Log Below Threshold , Log All and Halt CPU Above Threshold , and Log All and Halt CPU Below Threshold .

Context menu

This context menu is available:

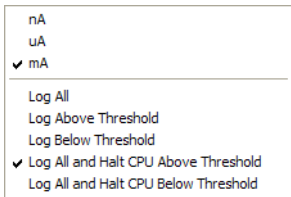


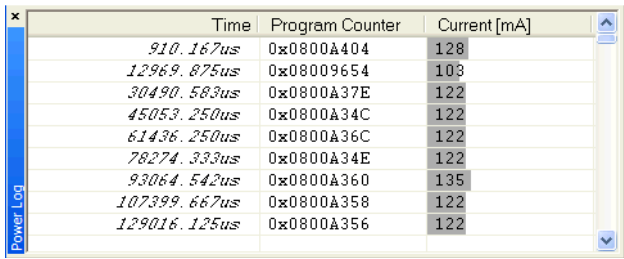
Figure 100: Power Setup window context menu

These commands are available:

nA, uA, mA	Selects the unit for the power display. These alternatives are available for channels that measure current.
Log All	Logs all values.
Log Above Threshold	Logs all values above the threshold.
Log Below Threshold	Logs all values below the threshold.
Log All and Halt CPU Above Threshold	Logs all values. If a logged value is above the threshold, execution is stopped.
Log All and Halt CPU Below Threshold	Logs all values. If a logged value is below the threshold, execution is stopped.

Power Log window

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



The screenshot shows the 'Power Log' window with a table containing three columns: Time, Program Counter, and Current [mA]. The 'Time' column contains values in italics, indicating they are approximations. The 'Current' column contains values in grey, indicating they are for channels that were active during data collection but are currently disabled. The table has 10 rows of data.

Time	Program Counter	Current [mA]
<i>910.167us</i>	0x0800A404	128
<i>12969.875us</i>	0x08009654	103
<i>30490.583us</i>	0x0800A37E	122
<i>45053.250us</i>	0x0800A34C	122
<i>61436.250us</i>	0x0800A36C	122
<i>78274.333us</i>	0x0800A34E	122
<i>93064.542us</i>	0x0800A360	135
<i>107399.667us</i>	0x0800A358	122
<i>129016.125us</i>	0x0800A356	122

Figure 101: Power Log window

This window is available for the J-Link driver.

This window displays collected power values.

A row with only the Time/Cycles and Program Counter displayed in grey denotes a logged power value for a channel that was active during the actual collection of data but currently is disabled in the Power Setup window.

Note: There is a limit in the number of logged power values. When this limit is exceeded, the entries in the beginning of the buffer are erased.

Display area

This area contains these columns:

Time	The time from the application reset until the event, based on the clock frequency specified in the SWO Configuration dialog box. If the time is displayed in italics, the target system could not collect a correct time, but instead had to approximate it. This column is available when you have selected Show Time from the context menu.
-------------	---

Cycles	<p>The number of cycles from the application reset until the event. This information is cleared at reset.</p> <p>If a cycle is displayed in italics, the target system could not collect a correct time, but instead had to approximate it.</p> <p>This column is available when you have selected Show Cycles from the context menu.</p>
Program Counter	<p>Displays one of these:</p> <p>An address, which is the content of the PC, that is, the address of an instruction in close to the point where the power value was collected.</p> <p>---, the target system failed to provide the debugger with any information.</p> <p>Overflow in red, the communication channel failed to transmit all data from the target system.</p> <p>Idle, the power value is logged during idle mode.</p>
Name [unit]	<p>The power measurement value expressed in the unit you specified in the Power Setup window.</p>

Context menu

This context menu is available:

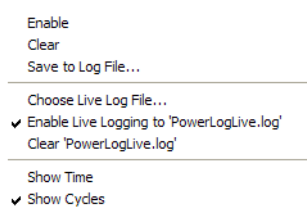


Figure 102: Power Log window context menu

These commands are available:

Enable	<p>Enables the logging system, which means power values are saved internally within the IDE. The values are displayed in the Power Log window and in the Power Graph in the Timeline window (if enabled). The system will log information also when the window is closed.</p>
--------	---

Clear	Clears the power values saved internally within the IDE. The values will also be cleared when you reset the debugger, or if you change the execution frequency in the SWO Setup dialog box.
Save to Log File	<p>Displays a standard file selection dialog box where you can choose the destination file for the logged power values. This command then saves the current content of the internal log buffer.</p> <p>For information about the file, see <i>The format of the log file</i>, page 238.</p>
Choose Live Log File	<p>Displays a standard file selection dialog box where you can choose a destination file for the logged power values. The power values are continuously saved to that file during execution. The content of the live log file is never automatically cleared, the logged values are simply appended at the end of the file.</p> <p>For information about the file, see <i>The format of the log file</i>, page 238.</p>
Enable Live Logging to	Toggles live logging on or off. The logs are saved in a to the specified file.
Clear log file	Clears the content of the live log file.
Show Time	Displays the Time column in the Power Log window. This choice is also reflected in the log files.
Show Cycles	Displays the Cycles column in the Power Log window. This choice is also reflected in the log files.

The format of the log file

The log file has a tab-separated format. The entries in the log file are separated by TAB and line feed. The logged power values are displayed in these columns:

Time/Cycles	The time from the application reset until the power value was logged.
Approx	An \times in the column indicates that the power value has an approximative value for time/cycle.
PC	The value of the program counter close to the point where the power value was logged.

Name[*unit*]

The corresponding value from the Power Log window, where *Name* and *unit* are according to your settings in the Power Setup window.

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 38. 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 242.

See also *Single stepping*, page 66.

Code Coverage window

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

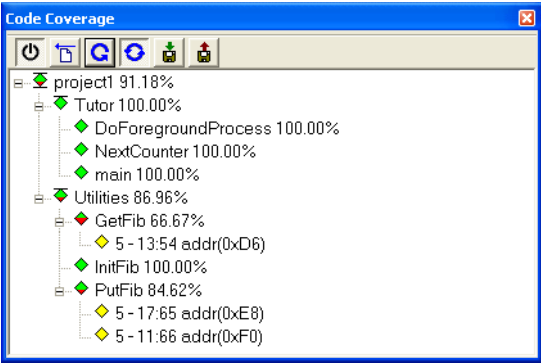


Figure 103: Code Coverage window

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

Table 16: Project options for enabling code coverage

Category	Setting
Linker	Output>Include debug information in output
Debugger	Plugins>Code Coverage

Table 16: 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:

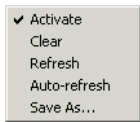


Figure 104: Code coverage window context menu

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. Interrupt logging provides you with comprehensive information about the interrupt events. 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
- Interrupt simulation states
- C-SPY system macros for interrupt simulation
- Target-adapting the interrupt simulation system
- Briefly about interrupt logging.

See also:

- *Reference information on C-SPY system macros*, page 281
- *Using breakpoints*, page 107
- *The IAR C/C++ Development Guide for ARM*.

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 ARM core
- 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. In addition, you can instantly force an interrupt.
- A log window that continuously displays events for each defined interrupt.
- A status window that shows the current interrupt activities.

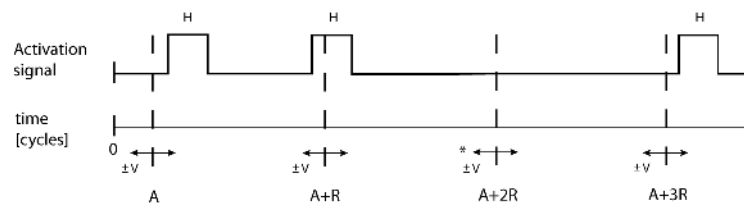
All interrupts you define using the **Interrupt Setup** dialog box are preserved between debug sessions, unless you remove them. A forced interrupt, on the other hand, exists only until it has been serviced and is not preserved between sessions.



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*.



* If probability is less than 100%, some interrupts may be omitted.

A = Activation time
R = Repeat interval
H = Hold time
V = Variance

Figure 105: 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. If the hold time is set to *infinite*, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

INTERRUPT SIMULATION STATES

The interrupt simulation system contains status information that you can use for locating timing problems in your application. The Interrupt Status window displays the available status information. For an interrupt, these states can be displayed: *Idle*, *Pending*, *Executing*, or *Suspended*.

Normally, a repeatable interrupt has a specified repeat interval that is longer than the execution time. In this case, the status information at different times looks like this:

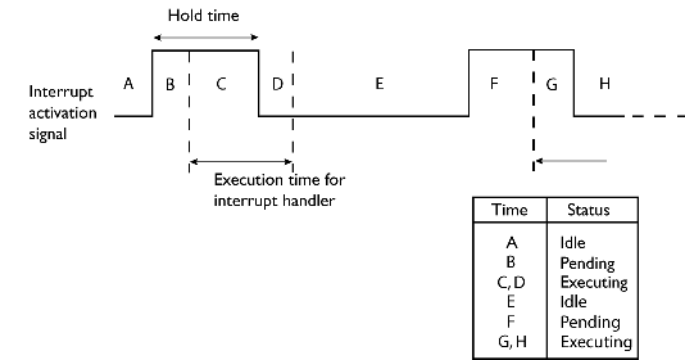


Figure 106: Simulation states - example 1

Note: The interrupt activation signal—also known as the pending bit—is automatically deactivated the moment the interrupt is acknowledged by the interrupt handler.

However, if the interrupt repeat interval is shorter than the execution time, and the interrupt is reentrant (or non-maskable), the status information at different times looks like this:

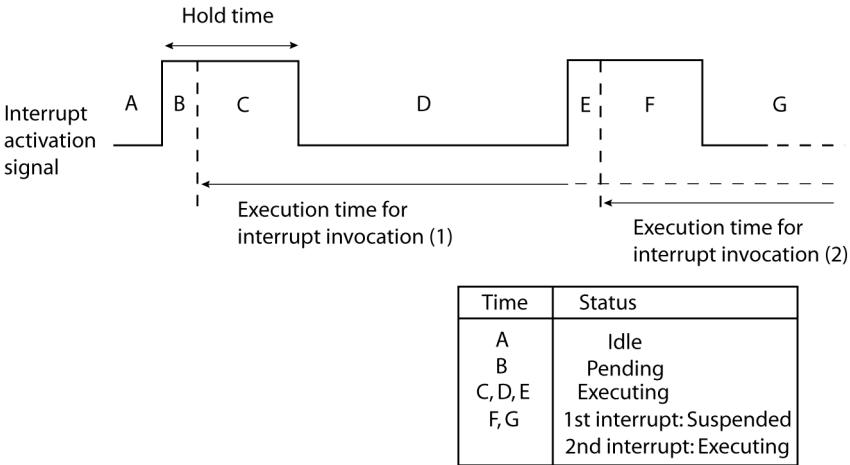


Figure 107: Simulation states - example 2

An execution time that is longer than the repeat interval might indicate that you should rewrite your interrupt handler and make it faster, or that you should specify a longer repeat interval for the interrupt simulation system.

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
__popSimulatorInterruptExecutingStack
```

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 281.

TARGET-ADAPTING THE INTERRUPT SIMULATION SYSTEM

The interrupt simulation system is easy to use. However, to take full advantage of the interrupt simulation system you should be familiar with how to adapt it for the processor you are using.

The interrupt simulation has the same behavior as the hardware. This means that the execution of an interrupt is dependent on the status of the global interrupt enable bit. The execution of maskable interrupts is also dependent on the status of the individual interrupt enable bits.

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 47.

BRIEFLY ABOUT INTERRUPT LOGGING

Interrupt logging provides you with comprehensive information about the interrupt events. This might be useful for example, to help you locate which interrupts you can fine-tune to become faster. You can log entrances and exits to and from interrupts. You can also log internal interrupt status information, such as triggered, expired, etc. The logs are displayed in the Interrupt Log window and a summary is available in the Interrupt Log Summary window. The Interrupt Graph in the Timeline window provides a graphical view of the interrupt events during the execution of your application program.

Requirements for interrupt logging

To use interrupt logging you need:

- A J-Link debug probe or an ST-LINK debug probe
- An SWD interface between the debug probe and the target system
- To enable interrupt logging from the Interrupt Log window, the Interrupt Log Summary window, or the Timeline window.

Interrupt logging is also supported by the C-SPY simulator.

Procedures for interrupts

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

More specifically, you will get information about:

- Simulating a simple interrupt
- Simulating an interrupt in a multi-task system
- Getting started using interrupt logging using C-SPY hardware drivers.

See also:

- *Registering and executing using setup macros and setup files*, page 272 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 for OKI ML674001. 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 IRQ handler routine that handles system timer interrupts. It increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```
/* Enables use of extended keywords */
#pragma language=extended

#include <intrinsics.h>
#include <oki/ioml674001.h>
#include <stdio.h>

unsigned int ticks = 0;

/* IRQ handler */
__irq __arm void IRQ_Handler(void)
{
    /* We use only system timer interrupts, so we do not need
       to check the interrupt source. */
    ticks += 1;
    TMOVFR_bit.OVF = 1; /* Clear system timer overflow flag */
}

int main( void )
{
    __enable_interrupt();
    /* Timer setup code */
    ILC0_bit.ILR0 = 4; /* System timer interrupt priority */
    TMRLR_bit.TMRLR = 1E5; /* System timer reload value */
    TMEN_bit.TCEN = 1; /* Enable system timer */
    while (ticks < 100);
    printf("Done\n");
}
```

- 2 Add your interrupt service routine to your application source code and add the file to your project.
- 3 Build your project and start the simulator.

- 4 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	IRQ
First activation	4000
Repeat interval	2000
Hold time	10
Probability (%)	100
Variance (%)	0

Table 17: Timer interrupt settings

Click **OK**.

- 5 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.
- 6 To watch the interrupt in action, choose **Simulator>Interrupt Log** to open the Interrupt Log window.
- 7 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 190.

SIMULATING AN INTERRUPT IN A MULTI-TASK SYSTEM

If you are using interrupts in such a way that the normal instruction used for returning from an interrupt handler is not used, for example in an operating system with task-switching, the simulator cannot automatically detect that the interrupt has finished executing. The interrupt simulation system will work correctly, but the status information in the **Interrupt Setup** dialog box might not look as you expect. If too many interrupts are executing simultaneously, a warning might be issued.

To simulate a normal interrupt exit:

- 1 Set a code breakpoint on the instruction that returns from the interrupt function.
- 2 Specify the `__popSimulatorInterruptExecutingStack` macro as a condition to the breakpoint.

When the breakpoint is triggered, the macro is executed and then the application continues to execute automatically.

GETTING STARTED USING INTERRUPT LOGGING USING C-SPY HARDWARE DRIVERS

Interrupt logging is supported by the J-Link/J-Trace and ST-LINK drivers.

- 1** To set up for interrupt logging, choose *C-SPY driver*>**SWO Configuration**. In the dialog box, set up the serial-wire output communication channel for trace data. Note specifically the **CPU clock** option. The CPU clock can also be set up on the **Project**>**Options**>**ST-LINK** page.
 - 2** Choose *C-SPY driver*>**Interrupt Log** to open the Interrupt Log window. Optionally, you can also choose:
 - *C-SPY driver*>**Interrupt Log Summary** to open the Interrupt Log Summary window
 - *C-SPY driver*>**Timeline** to open the Timeline window and view the Interrupt graph.
 - 3** From the context menu in the Interrupt Log window, choose **Enable** to enable the logging.
- In the **SWO Configuration** dialog box, you can see in the **Interrupt Log Events** area that interrupt logs are enabled.
- 4** Start executing your application program to collect the log information.
 - 5** To view the interrupt log information, look in any of the Interrupt Log, Interrupt Log Summary, or the Interrupt graph in the Timeline window.
 - 6** If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
 - 7** To disable interrupt logging, from the context menu in the Interrupt Log window, toggle **Enable** off.

Reference information on interrupts

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

- *Interrupt Setup dialog box*, page 254
- *Edit Interrupt dialog box*, page 256
- *Forced Interrupt window*, page 257
- *Interrupt Status window*, page 258
- *Interrupt Log window*, page 260

- *Interrupt Log Summary window*, page 264.

Interrupt Setup dialog box

The **Interrupt Setup** dialog box is available by choosing **Simulator>Interrupt Setup**.

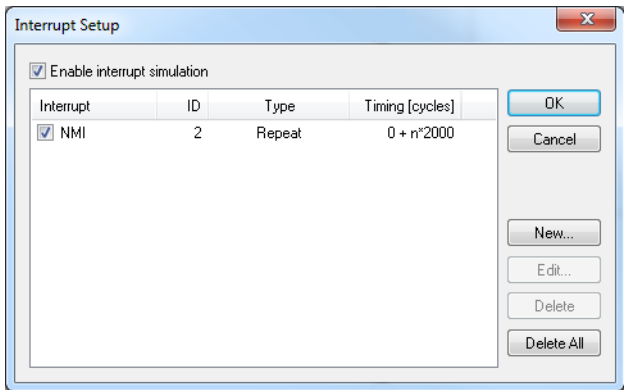


Figure 108: Interrupt Setup dialog box

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 interrupt simulation

Enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts are generated. Note that you can also enable and disable installed interrupts individually by using the check box to the left of the interrupt name in the list of installed interrupts.

Display area

This area contains these columns:

Interrupt	Lists all interrupts. Use the checkbox to enable or disable the interrupt.
ID	A unique interrupt identifier.

Type	<p>Shows the type of the interrupt. The type can be one of:</p> <p>Forced, a single-occasion interrupt defined in the Forced Interrupt Window.</p> <p>Single, a single-occasion interrupt.</p> <p>Repeat, a periodically occurring interrupt.</p> <p>If the interrupt has been set from a C-SPY macro, the additional part (macro) is added, for example: Repeat (macro).</p>
Timing	<p>The timing of the interrupt. For a Single and Forced interrupt, the activation time is displayed. For a Repeat interrupt, the information has the form: Activation Time + n*Repeat Time. For example, 2000 + n*2345. This means that the first time this interrupt is triggered, is at 2000 cycles and after that with an interval of 2345 cycles.</p>

Buttons

These buttons are available:

New	<p>Opens the Edit Interrupt dialog box, see <i>Edit Interrupt dialog box</i>, page 256.</p>
Edit	<p>Opens the Edit Interrupt dialog box, see <i>Edit Interrupt dialog box</i>, page 256.</p>
Delete	<p>Removes the selected interrupt.</p>
Delete All	<p>Removes all interrupts.</p>

Edit Interrupt dialog box

The **Edit Interrupt** dialog box is available from the **Interrupt Setup** dialog box.

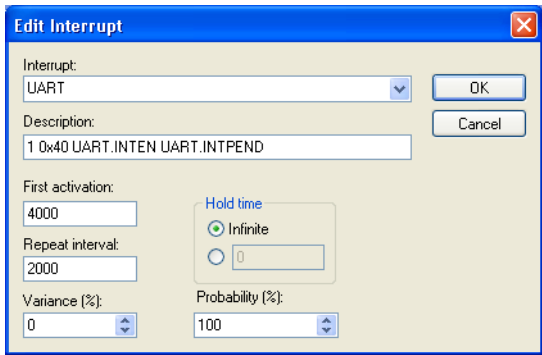


Figure 109: Edit Interrupt dialog box

Use this dialog box to interactively fine-tune the interrupt parameters. You can add the parameters and quickly test that the interrupt is generated according to your needs.

Note: You can only edit or remove non-forced interrupts.

Interrupt

Selects the interrupt that you want to edit. The drop-down list contains all available interrupts. Your selection will automatically update the **Description** box. For Cortex-M devices, the list is populated with entries from the device description file that you have selected. For other devices, only two interrupts are available: IRQ and FIQ.

Description

A description of the selected interrupt, if available. The description is retrieved from the selected device description file and consists of a string describing the priority, vector offset, enable bit, and pending bit, separated by space characters. The enable bit and pending bit are optional. It is possible to have none, only the enable bit, or both.

For Cortex-M devices, the description is retrieved from the selected device description file and is editable. Enable bit and pending bit are not available from the `adef` file; they must be manually edited if wanted. The priority is as in the hardware: the lower the number, the higher the priority. NMI and HardFault are special, and their descriptions should not be edited. Cortex-M interrupts are also affected by the `PRIMASK`, `FAULTMASK`, and `BASEPRI` registers, as described in the ARM documentation.

For other devices, the description strings for IRQ and FIQ are hardcoded and cannot be edited. In those descriptions, a higher priority number means a higher priority.

For interrupts specified using the system macro `__orderInterrupt`, the **Description** box is empty.

First activation

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.

Variance %

Selects a timing variation range, as a percentage of the repeat interval, in which the interrupt might 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.

Hold time

Specify how long, in cycles, the interrupt remains pending until removed if it has not been processed. If you select **Infinite**, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

Probability %

Selects the probability, in percent, that the interrupt will actually occur within the specified period.

Forced Interrupt window

The Forced Interrupt window is available from the **Simulator** menu.

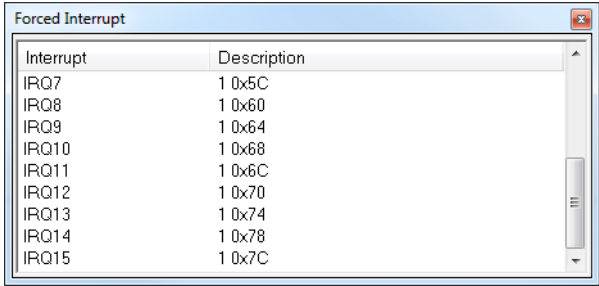


Figure 110: Forced Interrupt window

Use this window to force an interrupt instantly. This is useful when you want to check your interrupt logistics and interrupt routines.

The hold time for a forced interrupt is infinite, and the interrupt exists until it has been serviced or until a reset of the debug session.

To force an interrupt:

- 1 Enable the interrupt simulation system, see *Interrupt Setup dialog box*, page 254.
- 2 Double-click the interrupt in the Forced Interrupt window, or activate by using the **Force** command available on the context menu.

Display area

This area lists all available interrupts and their definitions. The information is retrieved from the selected device description file. See this file for a detailed description.

Context menu

This context menu is available:

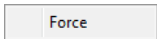


Figure 111: Forced Interrupt window context menu

This command is available:

Force Triggers the interrupt you selected in the display area.

Interrupt Status window

The Interrupt Status window is available from the **Simulator** menu.

Interrupt Status					
Interrupt	ID	Type	Status	Next Time	Timing [cycles]
NMI	0	Forced	Pending	--	--
IRQ0	1	Forced	Pending	--	--

Figure 112: Interrupt Status window

This window shows the status of all the currently active interrupts, in other words interrupts that are either executing or waiting to be executed.

Display area

This area contains these columns:

Interrupt	Lists all interrupts.
ID	A unique interrupt identifier.
Type	<p>The type of the interrupt. The type can be one of:</p> <p>Forced, a single-occasion interrupt defined in the Forced Interrupt window.</p> <p>Single, a single-occasion interrupt.</p> <p>Repeat, a periodically occurring interrupt.</p> <p>If the interrupt has been set from a C-SPY macro, the additional part (macro) is added, for example:</p> <p>Repeat (macro).</p>
Status	<p>The state of the interrupt:</p> <p>Idle, the interrupt activation signal is low (deactivated).</p> <p>Pending, the interrupt activation signal is active, but the interrupt has not been yet acknowledged by the interrupt handler.</p> <p>Executing, the interrupt is currently being serviced, that is the interrupt handler function is executing.</p> <p>Suspended, the interrupt is currently suspended due to execution of an interrupt with a higher priority.</p> <p>(deleted) is added to Executing and Suspended if you have deleted a currently active interrupt. (deleted) is removed when the interrupt has finished executing.</p>
Next Time	The next time an idle interrupt is triggered. Once a repeatable interrupt stats executing, a copy of the interrupt will appear with the state Idle and the next time set. For interrupts that do not have a next time—that is pending, executing, or suspended—the column will show --.

Timing

The timing of the interrupt. For a Single and Forced interrupt, the activation time is displayed. For a Repeat interrupt, the information has the form: Activation Time + n*Repeat Time. For example, 2000 + n*2345. This means that the first time this interrupt is triggered, is at 2000 cycles and after that with an interval of 2345 cycles.

Interrupt Log window

The Interrupt Log window is available from the **J-Link** menu, or the **ST-LINK** menu, respectively.

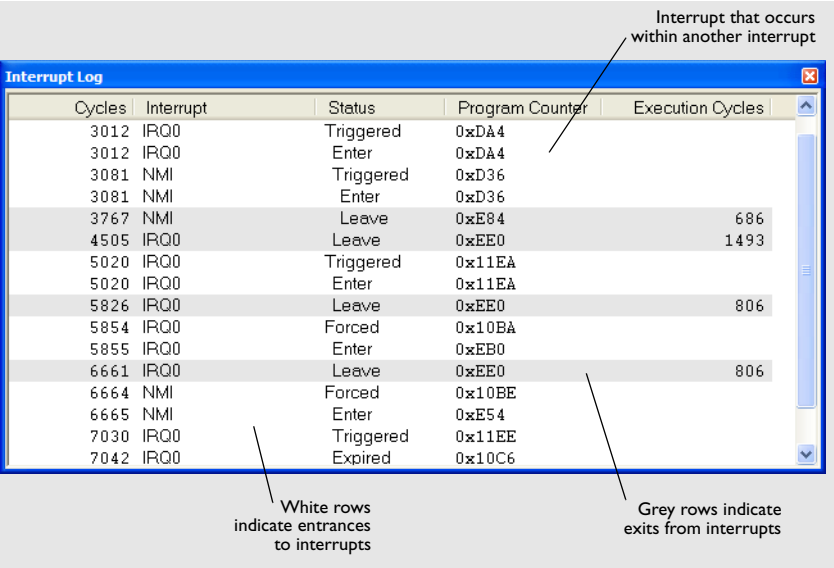


Figure 113: Interrupt Log window

To use the Interrupt Log window, you need one of these alternatives:

- A J-Link debug probe or an ST-LINK debug probe and an SWD interface between the debug probe and the target system.
- The C-SPY simulator.

This window logs entrances to and exits from interrupts. The C-SPY simulator also logs internal state changes.

The information is useful for debugging the interrupt handling in the target system. When the Interrupt Log window is open, it is updated continuously at runtime.

Note: There is a limit on the number of saved logs. When this limit is exceeded, the entries in the beginning of the buffer are erased.

For more information, see *Getting started using interrupt logging using C-SPY hardware drivers*, page 253.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 190.

Display area for the C-SPY J-Link driver and the ST-LINK driver

This area contains these columns:

Time	<p>The time for the interrupt entrance, based on the CPU clock frequency specified in the SWO Configuration dialog box.</p> <p>If a time is displayed in italics, the target system has not been able to collect a correct time, but instead had to approximate it.</p> <p>This column is available when you have selected Show Time from the context menu.</p>
Cycles	<p>The number of cycles from the start of the execution until the event.</p> <p>A cycle count displayed in italics indicates an approximative value. Italics is used when the target system has not been able to collect a correct value, but instead had to approximate it.</p> <p>This column is available when you have selected Show Cycles from the context menu.</p>
Interrupt	<p>The name of the interrupt source where the interrupt occurred. If the column displays <i>Overflow</i> in red, the communication channel failed to transmit all interrupt logs from the target system.</p>
Status	<p>The event status of the interrupt:</p> <p>Enter, the interrupt is currently executing.</p> <p>Leave, the interrupt has finished executing.</p>
Program Counter*	<p>The address of the interrupt handler.</p>

Execution Time/Cycles The time spent in the interrupt, calculated using the Enter and Leave timestamps. This includes time spent in any subroutines or other interrupts that occurred in the specific interrupt.

*** You can double-click an address. If it is available in the source code, the editor window displays the corresponding source code, for example for the interrupt handler (this does not include library source code).**

Display area for the C-SPY simulator

This area contains these columns:

Time	The time for the interrupt entrance, based on an internally specified clock frequency. This column is available when you have selected Show Time from the context menu.
Cycles	The number of cycles from the start of the execution until the event. This column is available when you have selected Show Cycles from the context menu.
Interrupt	The interrupt as defined in the device description file.
Status	Shows the event status of the interrupt: Triggered , the interrupt has passed its activation time. Forced , the same as Triggered, but the interrupt was forced from the Forced Interrupt window. Enter , the interrupt is currently executing. Leave , the interrupt has been executed. Expired , the interrupt hold time has expired without the interrupt being executed. Rejected , the interrupt has been rejected because the necessary interrupt registers were not set up to accept the interrupt.
Program Counter	The value of the program counter when the event occurred.
Execution Time/Cycles	The time spent in the interrupt, calculated using the Enter and Leave timestamps. This includes time spent in any subroutines or other interrupts that occurred in the specific interrupt.

Interrupt Log window context menu

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

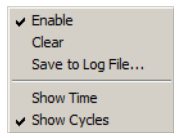


Figure 114: Interrupt Log window context menu

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

These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will happen also when you reset the debugger.
Save to log file	Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by <code>TAB</code> and <code>LF</code> . An <code>X</code> in the <code>Approx</code> column indicates that the timestamp is an approximation.
Show Time	Displays the Time column in the Data Log window and in the Interrupt Log window, respectively.
Show Cycles	Displays the Cycles column in the Data Log window and in the Interrupt Log window, respectively.

Interrupt Log Summary window

The Interrupt Log Summary window is available from the **J-Link** menu, or the **ST-LINK** menu, respectively.

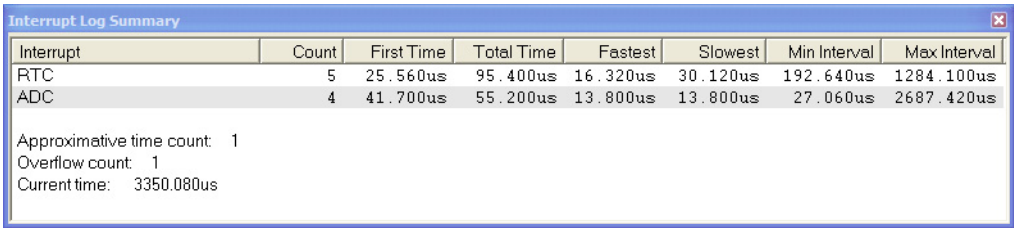


Figure 115: Interrupt Log Summary window

To use the Interrupt Log Summary window, you need one of these alternatives:

- A J-Link debug probe or an ST-LINK debug probe and an SWD interface between the debug probe and the target system.
- The C-SPY simulator.

This window displays a summary of logs of entrances to and exits from interrupts.

For more information, see *Getting started using interrupt logging using C-SPY hardware drivers*, page 253.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 190.

Display area for the C-SPY simulator

Each row in this area displays statistics about the specific interrupt based on the log information in these columns:

Interrupt*	The type of interrupt that occurred.
Count	The number of times the interrupt occurred.
First time	The first time the interrupt was executed.
Total time**	The accumulated time spent in the interrupt.
Fastest**	The fastest execution of a single interrupt of this type.
Slowest**	The slowest execution of a single interrupt of this type.
Max interval†	The longest time between two interrupts of this type.

* At the bottom of the column, the current time or cycles is displayed—the number of cycles or

the execution time since the start of execution. Overflow count and approximative time count is always zero.

** Calculated in the same way as for the Execution time/cycles in the Interrupt Log window.

† The interval is specified as the time interval between the entry time for two consecutive interrupts.

Context menu

See *Interrupt Log window context menu*, page 263.

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 `\arm\src\sim\`.

BRIEFLY ABOUT USING C-SPY MACROS

To use C-SPY macros, you should:

- Write your macro variables and functions and collect them in one or several *macro files*
- Register your macros
- Execute your macros.

For registering and executing macros, there are several methods to choose between. Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to register or execute your macro.

BRIEFLY ABOUT SETUP MACRO FUNCTIONS AND FILES

There are some reserved *setup macro function names* that you can use for defining macro functions which will be called at specific times, such as:

- Once after communication with the target system has been established but before downloading the application software
- Once after your application software has been downloaded
- Each time the reset command is issued
- Once when the debug session ends.

To define a macro function to be called at a specific stage, you should define and register a macro function with one of the reserved names. For instance, if you want to clear a specific memory area before you load your application software, the macro setup function `execUserPreload` should be used. This function is also suitable if you want to initialize some CPU registers or memory-mapped peripheral units before you load your application software.

You should define these functions in a *setup macro file*, which you can load before C-SPY starts. Your macro functions will then be automatically registered each time you start C-SPY. This is convenient if you want to automate the initialization of C-SPY, or if you want to register multiple setup macros.

For more information about each setup macro function, see *Reference information on reserved setup macro function names*, page 280.

Remapping memory

A common feature of many ARM-based processors is the ability to remap memory. After a reset, the memory controller typically maps address zero to non-volatile memory, such as flash. By configuring the memory controller, the system memory can be remapped to place RAM at zero and non-volatile memory higher up in the address map. By doing this, the exception table will reside in RAM and can be easily modified when you download code to the target hardware. To handle this in C-SPY, the setup macro function `execUserPreload()` is suitable. For an example, see *Remapping memory*, page 62.

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 275.

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 51.

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 271.
- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files*, page 272.
- 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 304.

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 272.

- 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 273.
- 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 274.

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**.

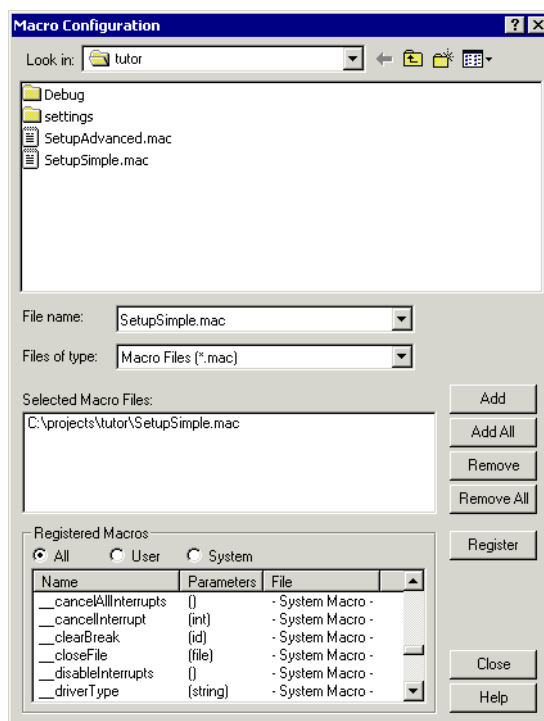


Figure 116: Macro Configuration dialog box

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:

- 1 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 (#WD_SR & 0x01 != 0) /* Checks the status of WDOVF */
        return "Watchdog triggered"; /* C-SPY macro string used */
    else
        return "Watchdog not triggered"; /* 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.

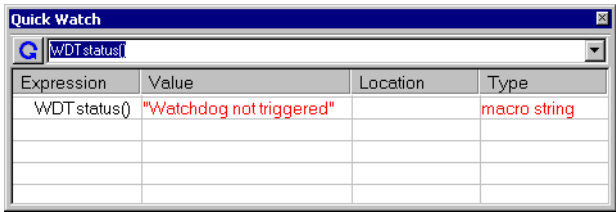


Figure 117: Quick Watch window

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

For more information, see *Quick Watch window*, page 98.

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 127
 - Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 118.
- 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 278.

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 276
- *Macro variables*, page 276
- *Macro strings*, page 277
- *Macro statements*, page 277
- *Formatted output*, page 278.

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 84.

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 18: 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 278.

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 84.

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 300.

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 268.

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>execUserFlashInit</code>	Called once before the flash loader is downloaded to RAM. Implement this macro typically for setting up the memory map required by the flash loader. This macro is only called when you are programming flash, and it should only be used for flash loader functionality.
<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>execUserFlashReset</code>	Called once after the flash loader is downloaded to RAM, but before execution of the flash loader. This macro is only called when you are programming flash, and it should only be used for flash loader functionality.
<code>execUserPreReset</code>	Called each time just before the reset command is issued. Implement this macro to set up any required device state.

Table 19: C-SPY setup macros

Macro	Description
<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.
<code>execUserFlashExit</code>	Called once when the debug session ends. Implement this macro to save status data etc. This macro is useful for flash loader functionality.

Table 19: C-SPY setup macros (Continued)



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>__emulatorSpeed</code>	Sets the emulator clock frequency
<code>__emulatorStatusCheckOnRead</code>	Enables or disables the verification of the CPSR register after each read operation
<code>__enableInterrupts</code>	Enables generation of interrupts
<code>__evaluate</code>	Interprets the input string as an expression and evaluates it.

Table 20: Summary of system macros

Macro	Description
__gdbserver_exec_command	Send strings or commands to the GDB Server.
__hwReset	Performs a hardware reset and halt of the target CPU
__hwResetRunToBp	Performs a hardware reset and then executes to the specified address
__hwResetWithStrategy	Performs a hardware reset and halt with delay of the target CPU
__isBatchMode	Checks if C-SPY is running in batch mode or not.
__jlinkExecCommand	Sends a low-level command to the J-Link/J-Trace driver.
__jtagCommand	Sends a low-level command to the JTAG instruction register
__jtagCP15IsPresent	Checks if coprocessor CP15 is available
__jtagCP15ReadReg	Returns the coprocessor CP15 register value
__jtagCP15WriteReg	Writes to the coprocessor CP15 register
__jtagData	Sends a low-level data value to the JTAG data register
__jtagRawRead	Returns the read data from the JTAG interface
__jtagRawSync	Writes accumulated data to the JTAG interface
__jtagRawWrite	Accumulates data to be transferred to the JTAG
__jtagResetTRST	Resets the ARM TAP controller via the TRST JTAG signal
__loadImage	Loads an image.
__memoryRestore	Restores the contents of a file to a specified memory zone
__memorySave	Saves the contents of a specified memory area to a file
__openFile	Opens a file for I/O operations
__orderInterrupt	Generates an interrupt
__popSimulatorInterruptExecutingStack	Informs the interrupt simulation system that an interrupt handler has finished executing
__readFile	Reads from the specified file
__readFileByte	Reads one byte from the specified file

Table 20: Summary of system macros (Continued)

Macro	Description
<code>__readMemory8,</code>	Reads one byte from the specified memory location
<code>__readMemoryByte</code>	
<code>__readMemory16</code>	Reads two bytes from the specified memory location
<code>__readMemory32</code>	Reads four bytes from the specified memory location
<code>__registerMacroFile</code>	Registers macros from the specified file
<code>__resetFile</code>	Rewinds a file opened by <code>__openFile</code>
<code>__restoreSoftwareBreakpoints</code>	Restores any breakpoints that were destroyed during system startup.
<code>__setCodeBreak</code>	Sets a code breakpoint
<code>__setDataBreak</code>	Sets a data breakpoint
<code>__setLogBreak</code>	Sets a log breakpoint
<code>__setSimBreak</code>	Sets a simulation breakpoint
<code>__setTraceStartBreak</code>	Sets a trace start breakpoint
<code>__setTraceStopBreak</code>	Sets a trace stop breakpoint
<code>__sourcePosition</code>	Returns the file name and source location if the current execution location corresponds to a source location
<code>__strFind</code>	Searches a given string for the occurrence of another string
<code>__subString</code>	Extracts a substring from another string
<code>__targetDebuggerVersion</code>	Returns the version of the target debugger
<code>__toLowerCase</code>	Returns a copy of the parameter string where all the characters have been converted to lower case
<code>__toString</code>	Prints strings
<code>__toUpperCase</code>	Returns a copy of the parameter string where all the characters have been converted to upper case
<code>__unloadImage</code>	Unloads a debug image.
<code>__writeFile</code>	Writes to the specified file
<code>__writeFileByte</code>	Writes one byte to the specified file

Table 20: Summary of system macros (Continued)

Macro	Description
<code>__writeMemory8,</code> <code>__writeMemoryByte</code>	Writes one byte to the specified memory location
<code>__writeMemory16</code>	Writes a two-byte word to the specified memory location
<code>__writeMemory32</code>	Writes a four-byte word to the specified memory location

Table 20: 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 21: `__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	<i>break_id</i>	The value returned by any of the set breakpoint macros
Return value	<code>int 0</code>	
Description	Clears a user-defined breakpoint.	
See also	<i>Using breakpoints</i> , page 107.	

__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 `__disableInterrupts()`

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 22: __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

<i>driver_id</i>	A string corresponding to the driver you want to check for. Choose one of these: "angel" corresponds to the Angel driver "gdbserv" corresponds to the GDB Server driver "generic" corresponds to third-party drivers "jlink" corresponds to the J-Link/J-Trace driver "jtag" corresponds to the Macraigor driver "lmiftdi" corresponds to the TI Stellaris driver "xds100" corresponds to the TI XDS100 driver "rdi" corresponds to the RDI driver "rom" corresponds to the IAR ROM-monitor driver "sim" corresponds to the simulator driver "stlink" corresponds to the ST-LINK driver.
------------------	---

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 23: __driverType return values

Description	Checks to see if the current C-SPY driver is identical to the driver type of the <i>driver_id</i> parameter.
Example	<pre>__driverType("sim")</pre> <p>If the simulator is the current driver, the value 1 is returned. Otherwise 0 is returned.</p>

__emulatorSpeed

Syntax	<pre>__emulatorSpeed(speed)</pre>
Parameter	<p><i>speed</i></p> <p>The emulator speed in Hz. Use 0 (zero) to make the speed automatically detected. Use -1 for adaptive speed (only for emulators supporting adaptive speed).</p>

Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td>The previous speed, or 0 (zero) if unknown</td></tr><tr><td>Unsuccessful; the speed is not supported by the emulator</td><td>-1</td></tr></table>	Result	Value	Successful	The previous speed, or 0 (zero) if unknown	Unsuccessful; the speed is not supported by the emulator	-1
Result	Value						
Successful	The previous speed, or 0 (zero) if unknown						
Unsuccessful; the speed is not supported by the emulator	-1						

Table 24: __emulatorSpeed return values

Description	Sets the emulator clock frequency. For JTAG interfaces, this is the JTAG clock frequency as seen on the TCK signal.
Applicability	This system macro is available for the J-Link/J-Trace driver.
Example	<pre>__emulatorSpeed(0)</pre> <p>Sets the emulator speed to be automatically detected.</p>

__emulatorStatusCheckOnRead

Syntax	<pre>__emulatorStatusCheckOnRead(status)</pre>
Parameter	<p><i>status</i></p> <p>Use 0 to enable checks (default). Use 1 to disable checks.</p>

Return value	int 0
Description	Enables or disables the driver verification of CPSR (current processor status register) after each read operation. Typically, this macro can be used for initiating JTAG connections on some CPUs, like Texas Instruments' TMS470R1B1M. Note: Enabling this verification can cause problems with some CPUs, for example if invalid CPSR values are returned. However, if this verification is disabled (SetCheckModeAfterRead = 0), the success of read operations cannot be verified and possible data aborts are not detected.
Applicability	This system macro is available for the J-Link/J-Trace driver.
Example	<pre>__emulatorStatusCheckOnRead(1)</pre> <p>Disables the checks for data aborts on memory reads.</p>

__enableInterrupts

Syntax	__enableInterrupts()							
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 25: __enableInterrupts return values								
Description	Enables the generation of interrupts.							
Applicability	This system macro is only available in the C-SPY Simulator.							

__evaluate

Syntax	<pre>__evaluate(string, valuePtr)</pre>	
Parameter	<i>string</i>	Expression string
	<i>valuePtr</i>	Pointer to a macro variable storing the result

Return value

Result	Value
Successful	int 0
Unsuccessful	int 1

Table 26: `__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.

`__gdbserver_exec_command`

Syntax

```
__gdbserver_exec_command("string")
```

Parameter

"string" String or command sent to the GDB Server; see its documentation for more information.

Description

Use this option to send strings or commands to the GDB Server.

Applicability

This system macro is available for the GDB Server driver.

`__hwReset`

Syntax

```
__hwReset(halt_delay)
```

Parameter

`halt_delay` The delay, in microseconds, between the end of the reset pulse and the halt of the CPU. Use 0 (zero) to make the CPU halt immediately after reset

Return value

Result	Value
Successful. The actual delay value implemented by the emulator	≥ 0
Successful. The delay feature is not supported by the emulator	-1

Table 27: `__hwReset` return values

	Result	Value
	Unsuccessful. Hardware reset is not supported by the emulator	-2

Table 27: `__hwReset` return values (Continued)

Description	Performs a hardware reset and halt of the target CPU.
Applicability	This system macro is available for all JTAG interfaces.
Example	<code>__hwReset(0)</code> Resets the CPU and immediately halts it.

`__hwResetRunToBp`

Syntax	<code>__hwResetRunToBp(strategy, breakpoint_address, timeout)</code>	
Parameter	<i>strategy</i>	For information about supported reset strategies, see the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> .
	<i>breakpoint_address</i>	The address of the breakpoint to execute to, specified as an integer value (symbols cannot be used).
	<i>timeout</i>	A time out for the breakpoint, specified in milliseconds. If the breakpoint is not reached within the specified time, the core will be halted.

Return value

Value	Result
<code>>=0</code>	Successful. The approximate execution time in ms until the breakpoint is hit.
<code>-2</code>	Unsuccessful. Hardware reset is not supported by the emulator.
<code>-3</code>	Unsuccessful. The reset strategy is not supported by the emulator.

Table 28: `__hwResetRunToBp` return values

Description	<p>Performs a hardware reset, sets a breakpoint at the specified address, executes to the breakpoint, and then removes it. The breakpoint address should be the start address of the downloaded image after it has been copied to RAM.</p> <p>This macro is intended for running a boot loader that copies the application image from flash to RAM. The macro should be executed after the image has been downloaded to</p>
-------------	---

flash, but before the image is verified. The macro can be run in `execUserFlashExit` or `execUserPreload`.

Applicability This system macro is available for the J-Link/J-Trace driver.

Example `__hwResetRunToBp(0, 0x400000, 10000)`
Resets the CPU with the reset strategy 0 and executes to the address 0x400000. If the breakpoint is not reached within 10 seconds, execution stops in accordance with the specified time out.

__hwResetWithStrategy

Syntax `__hwResetWithStrategy(halt_delay, strategy)`

Parameter

<i>halt_delay</i>	The delay, in milliseconds, between the end of the reset pulse and the halt of the CPU. Use 0 (zero) to make the CPU halt immediately after reset; only when <i>strategy</i> is set to 0.
<i>strategy</i>	For information about supported reset strategies, see the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> .

Return value	Result	Value
	Successful. The actual delay in milliseconds, as implemented by the emulator	=0
	Successful. The delay feature is not supported by the emulator	-1
	Unsuccessful. Hardware reset is not supported by the emulator	-2
	Unsuccessful. The reset strategy is not supported by the emulator	-3

Table 29: __hwResetWithStrategy return values

Description Performs a hardware reset and a halt with delay of the target CPU.

Applicability This system macro is available for the J-Link/J-Trace driver.

Example `__hwResetWithStrategy(0, 1)`
Resets the CPU and halts it using a breakpoint at memory address zero.

__isBatchMode

Syntax	<code>__isBatchMode()</code>							
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>True</td><td>int 1</td></tr><tr><td>False</td><td>int 0</td></tr></table>		Result	Value	True	int 1	False	int 0
Result	Value							
True	int 1							
False	int 0							
	<i>Table 30: <code>__isBatchMode</code> return values</i>							
Description	This macro returns True if the debugger is running in batch mode, otherwise it returns False.							

__jlinkExecCommand

Syntax	<code>__jlinkExecCommand(cmdstr)</code>	
Parameter	<i>cmdstr</i>	J-Link/J-Trace command string
Return value	int 0	
Description	Sends a low-level command to the J-Link/J-Trace driver. For a list of possible commands, see the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> .	
Applicability	This system macro is available for the J-Link/J-Trace driver.	
Example	See the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> .	
See also	<i>--jlink_exec_command</i> , page 338	

__jtagCommand

Syntax	<code>__jtagCommand(ir)</code>
--------	--------------------------------

Parameter	<p><i>ir</i> can be one of:</p> <table><tr><td>2</td><td>SCAN_N</td></tr><tr><td>4</td><td>RESTART</td></tr><tr><td>12</td><td>INTEST</td></tr><tr><td>14</td><td>IDCODE</td></tr><tr><td>15</td><td>BYPASS</td></tr></table>	2	SCAN_N	4	RESTART	12	INTEST	14	IDCODE	15	BYPASS
2	SCAN_N										
4	RESTART										
12	INTEST										
14	IDCODE										
15	BYPASS										
Return value	<code>int 0</code>										
Description	Sends a low-level command to the JTAG instruction register <code>IR</code> .										
Applicability	This system macro is available for the J-Link/J-Trace driver.										
Example	<pre>__jtagCommand(14); Id = __jtagData(0,32);</pre> <p>Returns the JTAG ID of the ARM target device.</p>										

__jtagCP15IsPresent

Syntax	<code>__jtagCP15IsPresent()</code>
Return value	1 if CP15 is available, otherwise 0.
Description	Checks if the coprocessor CP15 is available.
Applicability	This system macro is available for the J-Link/J-Trace driver.

__jtagCP15ReadReg

Syntax	<code>__jtagCP15ReadReg(CRn, CRm, op1, op2)</code>
Parameter	The parameters—registers and operands—of the MRC instruction. For details, see the <i>ARM Architecture Reference Manual</i> . Note that <i>op1</i> should always be 0.
Return value	The register value.
Description	Reads the value of the CP15 register and returns its value.

Applicability	This system macro is available for the J-Link/J-Trace driver.	
__jtagCP15WriteReg		
Syntax	<code>__jtagCP15WriteReg(CRn, CRm, op1, op2, value)</code>	
Parameter	The parameters—registers and operands—of the MCR instruction. For details, see the <i>ARM Architecture Reference Manual</i> . Note that <i>op1</i> should always be 0. <i>value</i> is the value to be written.	
Description	Writes a value to the CP15 register.	
Applicability	This system macro is available for the J-Link/J-Trace driver.	
__jtagData		
Syntax	<code>__jtagData(dr, bits)</code>	
Parameter	<i>dr</i>	32-bit data register value
	<i>bits</i>	Number of valid bits in <i>dr</i> , both for the macro parameter and the return value; starting with the least significant bit (1 . . . 32)
Return value	Returns the result of the operation; the number of bits in the result is given by the <i>bits</i> parameter.	
Description	Sends a low-level data value to the JTAG data register DR. The bit shifted out of DR is returned.	
Applicability	This system macro is available for the J-Link/J-Trace driver.	
Example	<pre>__jtagCommand(14); Id = __jtagData(0, 32);</pre> <p>Returns the JTAG ID of the ARM target device.</p>	

__jtagRawRead

Syntax	<code>__jtagRawRead(<i>bitpos</i>, <i>numbits</i>)</code>	
Parameter	<i>bitpos</i>	The start bit position in the returned JTAG bits to return data from
	<i>numbits</i>	The number of bits to read. The maximum value is 32.
Description	Returns the data read from the JTAG TDO. Only the least significant bits contain data; the last bit read is from the least significant bit. This function can be called an arbitrary number of times to get all bits returned by an operation. This function also makes an implicit synchronization of any accumulated write bits.	
Applicability	This system macro is available for the J-Link/J-Trace driver.	
Example	The following piece of pseudocode illustrates how the data is written to the JTAG (on the TMS and TDI pins) and read (from TDO):	

```
__var Id;
__var BitPos;
/*****
 *
 * ReadId()
 */
ReadId() {
    __message "Reading JTAG Id\n";
    __jtagRawWrite(0, 0x1f, 6); /* Goto IDLE via RESET state */
    __jtagRawWrite(0, 0x1, 3); /* Enter DR scan chain */
    BitPos = __jtagRawWrite(0, 0x80000000, 32); /* Shift 32 bits
                                                into DR. Remember BitPos for Read operation */
    __jtagRawWrite(0, 0x1, 2); /* Goto IDLE */
    Id = __jtagRawRead(BitPos, 32); /* Read the Id */
    __message "JTAG Id: ", Id:%x, "\n";
}
```

__jtagRawSync

Syntax	<code>__jtagRawSync()</code>
Return value	<code>int 0</code>

Description	Sends arbitrary data to the JTAG interface. All accumulated bits using <code>__jtagRawWrite</code> will be written to the JTAG scan chain. The data is sent synchronously with <code>TCK</code> and typically sampled by the device on rising edge of <code>TCK</code> .	
Applicability	This system macro is available for the J-Link/J-Trace driver.	
Example	<p>The following piece of pseudocode illustrates how the data is written to the JTAG (on the <code>TMS</code> and <code>TDI</code> pins) and read (from <code>TDO</code>):</p> <pre>int i; U32 tdo; for (i = 0; i < numBits; i++) { TDI = tdi & 1; /* Set TDI pin */ TMS = tms & 1; /* Set TMS pin */ TCK = 0; TCK = 1; tdo <<= 1; if (TDO) { tdo = 1; } tdi >>= 1; tms >>= 1; }</pre>	
<code>__jtagRawWrite</code>		
Syntax	<code>__jtagRawWrite(<i>tdi</i>, <i>tms</i>, <i>numbits</i>)</code>	
Parameter	<i>tdi</i>	The data output to the <code>TDI</code> pin. This data is sent with the least significant bit first.
	<i>tms</i>	The data output to the <code>TMS</code> pin. This data is sent with the least significant bit first.
	<i>numbits</i>	The number of bits to transfer. Every bit results in a falling and rising edge of the JTAG <code>TCK</code> line. The maximum value is 64.
Return value	Returns the bit position of the data in the accumulated packet. Typically, this value is used when reading data from the JTAG.	
Description	Accumulates bits to be transferred to the JTAG. If 32 bits are not enough, this function can be called multiple times. Both data output lines (<code>TMS</code> and <code>TDI</code>) can be controlled separately.	

Applicability This system macro is available for the J-Link/J-Trace driver.

Example

```
/* Send five 1 bits on TMS to go to TAP-RESET state */
__jtagRawWrite(0x1F, 0, 5); /* Store bits in buffer */
__jtagRawSync(); /* Transfer buffer, writing tms, tdi,
                  reading tdo */
```

Returns the JTAG ID of the ARM target device.

__jtagResetTRST

Syntax `__jtagResetTRST()`

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 31: `__jtagResetTRST` return values

Description Resets the ARM TAP controller via the TRST JTAG signal.

Applicability This system macro is available for the J-Link/J-Trace driver.

__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 for ARM</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	<table><tr><th>Value</th><th>Result</th></tr><tr><td>Non-zero integer number</td><td>A unique module identification.</td></tr><tr><td>int 0</td><td>Loading failed.</td></tr></table>	Value	Result	Non-zero integer number	A unique module identification.	int 0	Loading failed.
Value	Result						
Non-zero integer number	A unique module identification.						
int 0	Loading failed.						
Table 32: __loadImage return values							
Description	Loads an image (debug file).						
Example 1	<p>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 <code>execUserSetup</code> macro in a C-SPY macro file, which you associate with your project:</p> <pre>__loadImage(ROMfile, 0x8000, 1);</pre> <p>This macro call loads the debug information for the ROM library <i>ROMfile</i> without downloading its contents (because it is presumably already in ROM). Then you can debug your application together with the library.</p>						
Example 2	<p>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 <code>execUserSetup</code> macro in a C-SPY macro file, which you associate with your project:</p> <pre>__loadImage(ApplicationFile, 0x8000, 0);</pre> <p>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.</p>						
See also	<i>Images</i> , page 359 and <i>Loading multiple images</i> , page 50.						

__memoryRestore

Syntax	<code>__memoryRestore(zone, filename)</code>		
Parameters	<table><tr><td><i>zone</i></td><td>The memory zone name (string); for a list of available zones, see <i>C-SPY memory zones</i>, page 141.</td></tr></table>	<i>zone</i>	The memory zone name (string); for a list of available zones, see <i>C-SPY memory zones</i> , page 141.
<i>zone</i>	The memory zone name (string); for a list of available zones, see <i>C-SPY memory zones</i> , page 141.		

	<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 for ARM</i> .
Return value	int 0	
Description	Reads the contents of a file and saves it to the specified memory zone.	
Example	<pre>__memoryRestore("Memory", "c:\\temp\\saved_memory.hex");</pre>	
See also	<i>Memory Restore dialog box</i> , page 148.	

__memorySave

Syntax	<pre>__memorySave(start, stop, format, filename)</pre>	
Parameters	<p><i>start</i> A string that specifies the first location of the memory area to be saved</p> <p><i>stop</i> A string that specifies the last location of the memory area to be saved</p> <p><i>format</i> A string that specifies the format to be used for the saved memory. Choose between:</p> <pre>intel-extended motorola motorola-s19 motorola-s28 motorola-s37.</pre> <p><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 for ARM</i>.</p>	
Return value	int 0	
Description	Saves the contents of a specified memory area to a file.	
Example	<pre>__memorySave("Memory:0x00", "Memory:0xFF", "intel-extended", "c:\\temp\\saved_memory.hex");</pre>	
See also	<i>Memory Save dialog box</i> , page 147.	

__openFile

Syntax

`__openFile(filename, access)`

Parameters

filename The file to be opened. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide for ARM*.

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 33: __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 for ARM*.

__orderInterrupt**Syntax**

```
__orderInterrupt(specification, first_activation,
                 repeat_interval, variance, infinite_hold_time,
                 hold_time, probability)
```

Parameters

<i>specification</i>	The interrupt (string). The specification can either be the full specification used in the device description file (ddf) or only the name. In the latter case 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>infinite_hold_time</i>	1 if infinite, otherwise 0.
<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 *specification* 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:

```
__orderInterrupt( "IRQ", 4000, 2000, 0, 1, 0, 100 );
```

__popSimulatorInterruptExecutingStack

Syntax `__popSimulatorInterruptExecutingStack(void)`

Return value This macro has no return value.

Description Informs the interrupt simulation system that an interrupt handler has finished executing, as if the normal instruction used for returning from an interrupt handler was executed.

 This is useful if you are using interrupts in such a way that the normal instruction for returning from an interrupt handler is not used, for example in an operating system with task-switching. In this case, the interrupt simulation system cannot automatically detect that the interrupt has finished executing.

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

See also *Simulating an interrupt in a multi-task system*, page 252.

__readFile

Syntax `__readFile(fileHandle, valuePtr)`

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 34: __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

fileHandle A macro variable used as filehandle by the `__openFile` 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

address The memory address (integer)

zone The memory zone name (string); for more information about available zones, see *C-SPY memory zones*, page 141.

Return value

The macro returns the value from memory.

Description

Reads one byte from a given memory location.

Example

```
__readMemory8(0x0108, "Memory");
```

__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 141.
Return value	The macro returns the value from memory.	
Description	Reads a two-byte word from a given memory location.	
Example	<code>__readMemory16(0x0108, "Memory");</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 141.
Return value	The macro returns the value from memory.	
Description	Reads a four-byte word from a given memory location.	
Example	<code>__readMemory32(0x0108, "Memory");</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 for ARM</i> .

Return value	int 0
Description	Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.
Example	<pre>__registerMacroFile("c:\\testdir\\macro.mac");</pre>
See also	<i>Registering and executing using setup macros and setup files</i> , page 272.

__resetFile

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

__restoreSoftwareBreakpoints

Syntax	<pre>__restoreSoftwareBreakpoints()</pre>	
Return value	int 0	
Description	<p>Restores automatically any breakpoints that were destroyed during system startup.</p> <p>This can be useful if you have an application that is copied to RAM during startup and is then executing in RAM. This can, for example, be the case if you use the <code>initialize by copy</code> directive for code in the linker configuration file or if you have any <code>__ramfunc</code> declared functions in your application. In this case, any breakpoints will be overwritten during the RAM copying when the application execution starts.</p> <p>By using the this macro, C-SPY will restore the destroyed breakpoints.</p>	
Applicability	This system macro is available for the J-Link/J-Trace driver, the TI Stellaris driver, and the Macraigor driver.	

__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 84.</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 35: __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 107.

__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 84.

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)

<code>access</code>	The memory access type: "R" for read, "W" for write, or "RW" for read/write
<code>action</code>	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 36: `__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.

Example

```
__var brk;
brk = __setDataBreak("Memory:0x4710", 3, "d>6", "TRUE",
                    "W", "ActionData()");
...
__clearBreak(brk);
```

See also *Using breakpoints*, page 107.

__setLogBreak

Syntax

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

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 84.</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>message</i>	The message text
<i>msg_type</i>	<p>The message type; choose between:</p> <p>TEXT, the message is written word for word.</p> <p>ARGS, the message is interpreted as a comma-separated list of C-SPY expressions or strings.</p>
<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 37: __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	<pre> __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); } </pre>
See also	<i>Formatted output</i> , page 278 and <i>Using breakpoints</i> , page 107.

__setSimBreak

Syntax	__setSimBreak(<i>location</i> , <i>access</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 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 84.</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>	The memory access type: "R" for read or "W" for write
	<i>action</i>	An expression, typically a call to a macro function, 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 38: __setSimBreak return values

Description	<p>Use this system macro to set <i>immediate</i> 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.</p> <p>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.</p>
Applicability	<p>This system macro is only available in the C-SPY Simulator.</p>

__setTraceStartBreak

Syntax	<code>__setTraceStartBreak(location)</code>	
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 84.</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>

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 39: __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	<pre>__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); }</pre>
See also	<i>Using breakpoints</i> , page 107.

__setTraceStopBreak

Syntax	__setTraceStopBreak(<i>location</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 84.</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>

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 40: __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 312.
See also	Using breakpoints, page 107.

__sourcePosition

Syntax	__sourcePosition(<i>linePtr</i> , <i>colPtr</i>)	
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 41: __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(<i>macroString</i> , <i>pattern</i> , <i>position</i>)	
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	__strFind("Compiler", "pile", 0) = 3 __strFind("Compiler", "foo", 0) = -1	
See also	Macro strings, page 277.	

__subString

Syntax	<code>__subString(<i>macroString</i>, <i>position</i>, <i>length</i>)</code>	
Parameters	<i>macroString</i>	The macro string from which to extract a substring
	<i>position</i>	The start position of the substring. The first position is 0.
	<i>length</i>	The length of the substring
Return value	A substring extracted from the given macro string.	
Description	This macro extracts a substring from another string.	
Example	<code>__subString("Compiler", 0, 2)</code>	
	The resulting macro string contains Co.	
	<code>__subString("Compiler", 3, 4)</code>	
See also	The resulting macro string contains pile.	
	<i>Macro strings</i> , page 277.	

__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	<code>__var toolVer; toolVer = __targetDebuggerVersion(); __message "The target debugger version is, ", toolVer;</code>	

__toLower

Syntax	<code>__toLower(<i>macroString</i>)</code>	
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")</pre> <p>The resulting macro string contains <code>iar</code>.</p> <pre>__toLower("Mix42")</pre> <p>The resulting macro string contains <code>mix42</code>.</p>
See also	<i>Macro strings</i> , page 277.

__toString

Syntax	<code>__toString(C_string, maxlength)</code>	
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	<p>Assuming your application contains this definition:</p> <pre>char const * hpstr = "Hello World!";</pre> <p>this macro call:</p> <pre>__toString(hpstr, 5)</pre> <p>would return the macro string containing <code>Hello</code>.</p>	
See also	<i>Macro strings</i> , page 277.	

__toUpper

Syntax	<code>__toUpper(macroString)</code>
Parameter	<i>macroString</i> is any macro string.

Return value	The converted string.
Description	This macro returns a copy of the parameter <i>macroString</i> where all the characters have been converted to upper case.
Example	<pre>__toUpper("string")</pre> <p>The resulting macro string contains <i>STRING</i>.</p>
See also	<i>Macro strings</i> , page 277.

__unloadImage

Syntax	<code>__unloadImage(module_id)</code>							
Parameter	<i>module_id</i>	An integer which represents a unique module identification, which is retrieved as a return value from the corresponding <code>__loadImage</code> C-SPY macro.						
Return value	<table><tr><th>Value</th><th>Result</th></tr><tr><td><i>module_id</i></td><td>A unique module identification (the same as the input parameter).</td></tr><tr><td><code>int 0</code></td><td>The unloading failed.</td></tr></table>		Value	Result	<i>module_id</i>	A unique module identification (the same as the input parameter).	<code>int 0</code>	The unloading failed.
Value	Result							
<i>module_id</i>	A unique module identification (the same as the input parameter).							
<code>int 0</code>	The unloading failed.							

Table 42: `__unloadImage` return values

Description	Unloads debug information from an already downloaded image.
See also	<i>Loading multiple images</i> , page 50 and <i>Images</i> , page 359.

__writeFile

Syntax	<pre>__writeFile(fileHandle, value)</pre>	
Parameters	<i>fileHandle</i>	A macro variable used as filehandle by the <code>__openFile</code> macro
	<i>value</i>	An integer
Return value	int 0	

Description Prints the integer value in hexadecimal format (with a trailing space) to the file *file*.

Note: The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.

`__writeFileByte`

Syntax `__writeFileByte(fileHandle, value)`

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 `int 0`

Description Writes one byte to the file *fileHandle*.

`__writeMemory8`, `__writeMemoryByte`

Syntax `__writeMemory8(value, address, zone)`
`__writeMemoryByte(value, address, zone)`

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 141.

Return value `int 0`

Description Writes one byte to a given memory location.

Example `__writeMemory8(0x2F, 0x8020, "Memory");`

__writeMemory16

Syntax	<code>__writeMemory16(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 141.
Return value	<code>int 0</code>	
Description	Writes two bytes to a given memory location.	
Example	<code>__writeMemory16(0x2FFF, 0x8020, "Memory");</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 141.
Return value	<code>int 0</code>	
Description	Writes four bytes to a given memory location.	
Example	<code>__writeMemory32(0x5555FFFF, 0x8020, "Memory");</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>arm\bin</code> .
<i>driver_DLL</i>	The C-SPY driver DLL file; available in <code>arm\bin</code> .
<i>debug_file</i>	The object file that you want to debug (filename extension <code>.out</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 327.

Table 43: *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 327.

Table 43: cspybat parameters (Continued)

Example

This example starts cspybat using the simulator driver:

```
EW_DIR\common\bin\cspybat EW_DIR\arm\bin\armproc.dll
EW_DIR\arm\bin\armsim.dll PROJ_DIR\myproject.out --plugin
EW_DIR\arm\bin\armbat.dll --backend sim -B --cpu arm -p
EW_DIR\arm\bin\config\devicedescription.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 346.
- 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

GENERAL CSPYBAT OPTIONS

<code>--backend</code>	Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).
<code>--code_coverage_file</code>	Enables the generation of code coverage information and places it in a specified file.
<code>--cycles</code>	Specifies the maximum number of cycles to run.
<code>--download_only</code>	Downloads a code image without starting a debug session afterwards.
<code>--flash_loader</code>	Specifies a flash loader specification XML file.
<code>--macro</code>	Specifies a macro file to be used.
<code>--plugin</code>	Specifies a plugin file to be used.
<code>--silent</code>	Omits the sign-on message.
<code>--timeout</code>	Limits the maximum allowed execution time.

OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

<code>--BE8</code>	Uses the big-endian format BE8. For reference information, see the <i>IAR C/C++ Development Guide for ARM</i> .
<code>--BE32</code>	Uses the big-endian format BE32. For reference information, see the <i>IAR C/C++ Development Guide for ARM</i> .

<code>--cpu</code>	Specifies a processor variant. For reference information, see the <i>IAR C/C++ Development Guide for ARM</i> .
<code>--device</code>	Specifies the name of the device.
<code>--drv_attach_to_program</code>	Attaches the debugger to a running application at its current location. For reference information, see <i>Attach to program</i> , page 357.
<code>--drv_catch_exceptions</code>	Makes the application stop for certain exceptions.
<code>--drv_communication</code>	Specifies the communication link to be used.
<code>--drv_communication_log</code>	Creates a log file.
<code>--drv_default_breakpoint</code>	Sets the type of breakpoint resource to be used when setting breakpoints.
<code>--drv_reset_to_cpu_start</code>	Omits setting the PC when starting or resetting the debugger.
<code>--drv_restore_breakpoints</code>	Restores automatically any breakpoints that were destroyed during system startup.
<code>--drv_suppress_download</code>	Suppresses download of the executable image. For reference information, see <i>Suppress download</i> , page 357.
<code>--drv_swo_clock_setup</code>	Specifies the CPU clock and the wanted SWO speed.
<code>--drv_vector_table_base</code>	Specifies the location of the Cortex-M reset vector and the initial stack pointer value.
<code>--drv_verify_download</code>	Verifies the target program. For reference information, see <i>Verify download</i> , page 357. Available for Angel, GDB Server, IAR ROM-monitor, J-Link/J-Trace, JTAGjet, Macraigor, RDI, ST-LINK, TI Stellaris, and TI XDS100.
<code>--endian</code>	Specifies the byte order of the generated code and data. For reference information, see the <i>IAR C/C++ Development Guide for ARM</i> .

<code>--fpu</code>	Selects the type of floating-point unit. For reference information, see the <i>IAR C/C++ Development Guide for ARM</i> .
<code>-p</code>	Specifies the device description file to be used.
<code>--proc_stack_stack</code>	Provides information to the C-SPY plugin module about reserved stacks.
<code>--semihosting</code>	Enables semihosted I/O.

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

<code>--disable_interrupts</code>	Disables the interrupt simulation.
<code>--mapu</code>	Activates memory access checking.

OPTIONS AVAILABLE FOR THE C-SPY ANGEL DEBUG MONITOR DRIVER

<code>--rdi_heartbeat</code>	Makes C-SPY poll your target system periodically. For reference information, see <i>Send heartbeat</i> , page 362.
<code>--rdi_step_max_one</code>	Executes one instruction.

OPTIONS AVAILABLE FOR THE C-SPY GDB SERVER DRIVER

<code>--gdbserv_exec_command</code>	Sends a command string to the GDB Server.
-------------------------------------	---

OPTIONS AVAILABLE FOR THE C-SPY IAR ROM-MONITOR DRIVER

There are no additional options specific to the C-SPY IAR ROM-monitor driver.

OPTIONS AVAILABLE FOR THE C-SPY J-LINK/J-TRACE DRIVER

<code>--jlink_device_select</code>	Selects a specific device in the JTAG scan chain.
<code>--jlink_exec_command</code>	Calls the <code>__jlinkExecCommand</code> macro after target connection has been established.
<code>--jlink_initial_speed</code>	Sets the initial JTAG communication speed in kHz.

<code>--jlink_interface</code>	Specifies the communication between the J-Link debug probe and the target system.
<code>--jlink_ir_length</code>	Sets the number of IR bits before the ARM device to be debugged.
<code>--jlink_reset_strategy</code>	Selects the reset strategy to be used at debugger startup.
<code>--jlink_script_file</code>	Specifies the script file for setting up hardware.
<code>--jlink_speed</code>	Sets the JTAG communication speed in kHz.
<code>--jlink_trace_source</code>	Selects either ETB or ETM as the trace source.

OPTIONS AVAILABLE FOR THE C-SPY TI STELLARIS DRIVER

<code>--lmiftdi_speed</code>	Sets the JTAG communication speed in kHz.
------------------------------	---

OPTIONS AVAILABLE FOR THE C-SPY TI XDS100 DRIVER

<code>--xds_rootdir</code>	Specifies the installation directory of the TI XDS100 driver package.
----------------------------	---

OPTIONS AVAILABLE FOR THE C-SPY MACRAIGOR DRIVER

<code>--mac_handler_address</code>	Specifies the location of the debug handler used by Intel XScale devices.
<code>--mac_interface</code>	Specifies the communication between the Macraigor debug probe and the target system.
<code>--mac_jtag_device</code>	Selects the device corresponding to the hardware interface.
<code>--mac_multiple_targets</code>	Specifies the device to connect to, if there are more than one device on the JTAG scan chain.
<code>--mac_reset_pulls_reset</code>	Makes C-SPY generate an initial hardware reset.
<code>--mac_set_temp_reg_buffer</code>	Provides the driver with a physical RAM address for accessing the coprocessor.
<code>--mac_speed</code>	Sets the JTAG speed between the JTAG probe and the ARM JTAG ICE port.

<code>--mac_xscale_ir7</code>	Specifies that the XScale ir7 architecture is used.
-------------------------------	---

OPTIONS AVAILABLE FOR THE C-SPY RDI DRIVER AND THE JTAGJET DRIVER

<code>--rdi_allow_hardware_reset</code>	Performs a hardware reset.
<code>--rdi_driver_dll</code>	Specifies the path to the driver DLL file.
<code>--rdi_step_max_one</code>	Executes one instruction.

OPTIONS AVAILABLE FOR THE C-SPY ST-LINK DRIVER

<code>--stlink_interface</code>	Specifies the communication between the ST-LINK debug probe and the target system.
<code>--stlink_reset_strategy</code>	Specifies the reset strategy to use.

OPTIONS AVAILABLE FOR THE C-SPY THIRD-PARTY DRIVERS

For information about any options specific to the third-party driver you are using, see its documentation.

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.

--backend

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

--code_coverage_file


Syntax	<code>--code_coverage_file file</code>	
Parameters	<i>file</i>	The name of the destination file for the code coverage information.
Applicability	Sent to <i>cspybat</i> .	
Description	<p>Use this option to enable the generation of code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file.</p> <p>Note that this option requires that the C-SPY driver you are using supports code coverage. If you try to use this option with a C-SPY driver that does not support code coverage, an error message will be directed to <i>stderr</i>.</p>	
See also	<i>Code coverage</i> , page 241.	

--cycles


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

--device


Syntax	<code>--device=device_name</code>	
Parameters	<i>device_name</i>	The name of the device, for example, ADuC7030, AT91SAM7S256, LPC2378, STR912FM44, or TMS470R1B1M.

Applicability	All C-SPY drivers.
Description	Use this option to specify the name of the device. To set related option, choose:  Project>Options>General Options>Target>Device


--disable_interrupts

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

--drv_catch_exceptions

Syntax	<code>--drv_catch_exceptions=value</code>	
Parameters	<i>value</i> (for ARM9 and Cortex-R4)	A value in the range of 0–0x1FF. Each bit specifies which exception to catch: Bit 0 = Reset Bit 1 = Undefined instruction Bit 2 = SWI Bit 3 = Not used Bit 4 = Data abort Bit 5 = Prefetch abort Bit 6 = IRQ Bit 7 = FIQ Bit 8 = Other errors
	<i>value</i> (for Cortex-M)	A value in the range of 0–0x7FF. Each bit specifies which exception to catch: Bit 0 = CORERESSET - Reset Vector Bit 4 = MMERR - Memory Management Fault Bit 5 = NOCPERR - Coprocessor Access Error Bit 6 = CHKERR - Checking Error Bit 7 = STATERR - State Error Bit 8 = BUSERR - Bus Error Bit 9 = INTERR - Interrupt Service Errors Bit 10 = HARDERR - Hard Fault
Applicability	The C-SPY Angel debug monitor driver. The C-SPY J-Link/J-Trace driver The C-SPY JTAGjet driver The C-SPY RDI driver.	
Description	Use this option to make the application stop when a certain exception occurs.	
See also	<i>Breakpoints on exception vectors</i> , page 112.	
		For the C-SPY Angel debug monitor driver, use: Project>Options>Debugger>Extra Options

For the C-SPY J-Link/J-Trace driver, use:

Project>Options>Debugger>J-Link/J-Trace>Catch exceptions

For the C-SPY RDI driver, use:

Project>Options>Debugger>RDI>Catch exceptions

--drv_communication

Syntax

--drv_communication=*connection*

Parameters

Where *connection* is one of these for the C-SPY Angel debug monitor driver:

Via Ethernet

UDP:*ip_address*
 UDP:*ip_address,port*
 UDP:*hostname*
 UDP:*hostname,port*

Via serial port

port:baud,parity,stop_bit,handshake
port = COM1-COM256 (default COM1)
baud = 9600, 19200, 38400, 57600, or 115200 (default 9600 baud)
parity = N (no parity)
stop_bit = 1 (one stop bit)
handshake = NONE or RTSCTS (default NONE for no handshaking)

For example, COM1:9600,N,8,1,NONE.

Where *connection* is one of these for the C-SPY GDB Server driver:

Via Ethernet

TCP/IP:*ip_address*
 TCP/IP:*ip_address,port*
 TCP/IP:*hostname*
 TCP/IP:*hostname,port*

Note that if no port is specified, port 3333 is used by default.

Where *connection* is one of these for the C-SPY IAR ROM-monitor driver:

Via serial port *port:baud,parity,stop_bit,handshake*
 port = COM1-COM256 (default COM1)
 baud = 9600, 19200, 38400, 57600, or 115200 (default
 9600 baud)
 parity = N (no parity)
 stop_bit = 1 (one stop bit)
 handshake = NONE or RTSCTS (default NONE for no
 handshaking)
 For example, COM1:9600,N,8,1,NONE.

Where *connection* is one of these for the C-SPY J-Link/J-Trace driver:

Via USB directly to USB0-USB3
 J-Link When using USB0 and if there are more than one J-Link debug
 probes on the USB connection, a dialog box is displayed when
 the debug session starts. Use the dialog box to choose which
 J-Link debug probe to connect to.
 USB:#*number*, connects to the J-Link with the serial number
 number on the USB connection

Via J-Link on LAN TCPIP:
 When the colon sign is not followed by any address, host
 name, or serial number, the J-Link driver searches for all
 J-Link debug probes on the local network and displays
 them in a dialog box where you can choose which one to
 connect to (Auto detect).
 TCPIP:*ip_address*
 TCPIP:*ip_address,port*
 TCPIP:*hostname*
 TCPIP:*hostname,port*
 TCPIP:#*number*, connects to the J-Link with the serial
 number *number* on the local network
 Note that if no port is specified, port 19020 is used by default.

Where *connection* is one of these for the C-SPY Macraigor driver:

For mpDemon *port:baud*
 port = COM1–COM4
 baud = 9600, 19200, 38400, 57600, or 115200 (default
 9600 baud)

For mpDemon TCPIP:*ip_address*
 TCPIP:*ip_address,port*
 TCPIP:*hostname*
 TCPIP:*hostname,port*

Note that if no port is specified, port 19020 is used by default.

Via USB to usbDemon USB ports = USB0–USB3
 and usb2Demon

Applicability

The C-SPY Angel debug monitor driver
 The C-SPY GDB Server driver
 The C-SPY IAR ROM-monitor driver
 The C-SPY J-Link/J-Trace driver
 The C-SPY Macraigor driver.

Description

Use this option to choose communication link.



Project>Options>Debugger>Angel>Communication

Project>Options>Debugger>GDB Server>TCP/IP address or hostname [,port]

Project>Options>Debugger>IAR ROM-monitor>Communication

Project>Options>Debugger>J-Link/J-Trace>Connection>Communication

To set related options for the C-SPY Macraigor driver, choose:

Project>Options>Debugger>Macraigor

--drv_communication_log

Syntax

`--drv_communication_log=filename`

Parameters

filename The name of the log file.

Applicability	All C-SPY 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>Driver>Log communication

--drv_default_breakpoint


Syntax	<code>--drv_default_breakpoint={0 1 2}</code>	
Parameters	0	Auto (default)
	1	Hardware
	2	Software
Applicability	The C-SPY GDB Server driver	
	The C-SPY J-Link/J-Trace driver	
	The C-SPY Macraigor driver.	
Description	Use this option to select the type of breakpoint resource to be used when setting a breakpoint.	
See also	<i>Default breakpoint type</i> , page 133.	




Project>Options>Debugger>Driver>Breakpoints>Default breakpoint type

--drv_reset_to_cpu_start

Syntax	<code>--drv_reset_to_cpu_start</code>
Applicability	The C-SPY Angel debug monitor driver
	The C-SPY GDB Server driver
	The C-SPY J-Link/J-Trace driver
	The C-SPY TI Stellaris driver
	The C-SPY TI XDS100 driver
	The C-SPY Macraigor driver


	The C-SPY JTAGjet driver
	The C-SPY RDI driver
	The C-SPY ST-LINK driver.
Description	Use this option to omit setting the PC when starting or resetting the debugger. Instead PC will have the original value set by the CPU, which is the address of the application entry point.
	 To set this option, use Project>Options>Debugger>Extra Options .

--drv_restore_breakpoints

Syntax	<code>--drv_restore_breakpoints=location</code>	
Parameters	<i>location</i>	Address or function name label
Applicability	The C-SPY GDB Server driver The C-SPY J-Link/J-Trace driver The C-SPY Macraigor driver.	
Description	Use this option to restore automatically any software breakpoints that were overwritten during system startup.	
See also	<i>Restore software breakpoints at</i> , page 133.	
		Project>Options>Debugger>Driver>Breakpoints>Restore software breakpoints at

--drv_swo_clock_setup

Syntax	<code>--drv_swo_clock_setup=frequency, autodetect, wanted</code>	
Parameters	<i>frequency</i>	The exact clock frequency used by the internal processor clock, HCLK, in Hz. This value is used for configuring the SWO communication speed and for calculating timestamps.

	<i>autodetect</i>	0, Specify the wanted frequency using the parameter <i>wanted</i> . 1, Automatically uses the highest possible frequency that the J-Link debug probe can handle.
	<i>wanted</i>	The frequency to be used, if <i>autodetect</i> is 0, in Hz. Use <i>wanted</i> if data packets are lost during transmission.
Applicability	The J-Link driver and the ST-LINK driver.	
Description	Use this option to set up the CPU clock. If this option is not used, the CPU clock frequency is by default set to 72 MHz.	
		J-Link>SWO Configuration>CPU clock J-Link>SWO Configuration>SWO clock>Autodetect J-Link>SWO Configuration>SWO clock>Wanted
--drv_vector_table_base		
Syntax	<code>--drv_vector_table_base=expression</code>	
Parameters	<i>expression</i>	A label or an address
Applicability	The C-SPY GDB Server driver The C-SPY J-Link/J-Trace driver The C-SPY TI Stellaris driver The C-SPY TI XDS100 driver The C-SPY Macraigor driver The C-SPY JTAGjet driver The C-SPY RDI driver The C-SPY ST-LINK driver The C-SPY Simulator driver.	
Description	Use this option for Cortex-M to specify the location of the reset vector and the initial stack pointer value. This is useful if you want to override the default <code>__vector_table</code> label—defined in the system startup code—in the application or if the application lacks	

this label, which can be the case if you debug code that is built by tools from another vendor.



To set this option, use **Project>Options>Debugger>Extra Options**.

--flash_loader

Syntax	<code>--flash_loader filename</code>	
Parameters	<i>filename</i>	The flash loader specification XML file.
Applicability	Sent to cspybat.	
Description	Use this option to specify a flash loader specification xml file which contains all relevant information about the flash loading. There can be more than one such argument, in which case each argument will be processed in the specified order, resulting in several flash programming passes.	
See also	The <i>IAR Embedded Workbench flash loader User Guide</i> .	

--gdbserv_exec_command


Syntax	<code>--gdbserv_exec_command="string"</code>	
Parameters	<i>"string"</i>	String or command sent to the GDB Server; see its documentation for more information.
Applicability	The C-SPY GDB Server driver.	
Description	Use this option to send strings or commands to the GDB Server.	




Project>Options>Debugger>Extra Options

--jlink_device_select

Syntax	<code>--jlink_device_select=tap_number</code>
--------	---

Parameters	<i>tap_number</i>	The TAP position of the device you want to connect to.
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	If there is more than one device on the JTAG scan chain, use this option to select a specific device.	
See also	<i>JTAG scan chain</i> , page 371.	
		Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>TAP number

--jlink_exec_command

Syntax	<code>--jlink_exec_command=cmdstr1; cmdstr2; cmdstr3 ...</code>	
Parameters	<i>cmdstrn</i>	J-Link/J-Trace command string.
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to make the debugger call the <code>__jlinkExecCommand</code> macro with one or several command strings, after target connection has been established.	
See also	<i>__jlinkExecCommand</i> , page 292.	
		Project>Options>Debugger>Extra Options

--jlink_initial_speed

Syntax	<code>--jlink_initial_speed=speed</code>	
Parameters	<i>speed</i>	The initial communication speed in kHz. If no speed is specified, 32 kHz will be used as the initial speed.
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to set the initial JTAG communication speed in kHz.	

See also *JTAG/SWD speed*, page 368.



Project>Options>Debugger>J-Link/J-Trace>Setup>JTAG speed>Fixed

--jlink_interface

Syntax	<code>--jlink_interface={JTAG SWD}</code>	
Parameters	JTAG	Uses JTAG communication with the target system (default).
	SWD	Uses SWD communication with the target system (Cortex-M only); uses fewer pins than JTAG communication.
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to specify the communication channel between the J-Link debug probe and the target system.	
See also	<i>Interface</i> , page 370.	



Project>Options>Debugger>J-Link/J-Trace>Connection>Interface

--jlink_ir_length

Syntax	<code>--jlink_ir_length=length</code>	
Parameters	<i>length</i>	The number of IR bits before the ARM device to be debugged, for JTAG scan chains that mix ARM devices with other devices.
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to set the number of IR bits before the ARM device to debugged.	
See also	<i>JTAG scan chain</i> , page 371.	



Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>Preceding bits

--jlink_reset_strategy

Syntax	<code>--jlink_reset_strategy=delay, strategy</code>	
Parameters	<i>delay</i>	For Cortex-M and ARM 7/9/11 with strategies 1-9, <i>delay</i> should be 0 (ignored). For ARM 7/9/11 with strategy 0, the delay should be one of 0-10000.
	<i>strategy</i>	For information about supported reset strategies, see the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> .
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to select the reset strategy to be used at debugger startup.	
See also	<i>Reset</i> , page 365.	



Project>Options>Debugger>J-Link/J-Trace>Setup>Reset

--jlink_script_file

Syntax	<code>--jlink_script_file=filename</code>	
Parameters	<i>filename</i>	The name of the J-Link script file.
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to specify the J-Link script file to be used.	
	J-Link has a script language that can be used for setting up hardware. For certain targets, ready-made script files are automatically pointed out by IAR Embedded Workbench. In command line mode, the script file needs to be manually specified by using this option.	
	When using a non-predefined script file, this option can be passed to C-SPY on the Project>Options>Debugger>Extra Options page.	
See also	The <i>J-Link/J-Trace ARM User Guide</i> (JLinkARM.pdf , document number UM08001), section 5.10, for a detailed description of the script language.	

--jlink_speed

Syntax	--jlink_speed={fixed auto adaptive}	
Parameters	fixed	1-12000
	auto	The highest possible frequency for reliable operation (default)
	adaptive	For ARM devices that have the RTCK JTAG signal available
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to set the JTAG communication speed in kHz.	
See also	JTAG/SWD speed, page 381.	



Project>Options>Debugger>J-Link/J-Trace>Setup>JTAG speed

--jlink_trace_source

Syntax	--jlink_trace_source={ETB ETM}	
Parameters	ETB	Selects ETB trace.
	ETM	Selects ETM trace.
Applicability	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to select either ETB or ETM as the trace source. Note: This option applies only to J-Trace.	
See also	ETM/ETB, page 369.	



Project>Options>Debugger>J-Link/J-Trace>Setup>ETM/ETB

--lmiftdi_speed

Syntax	<code>--lmiftdi_speed=frequency</code>	
Parameters	<i>frequency</i>	The frequency in kHz.
Applicability	The C-SPY TI Stellaris driver.	
Description	Use this option to set the JTAG communication speed in kHz.	
See also	<i>JTAG/SWD speed</i> , page 381.	



Project>Options>Debugger>TI Stellaris>Setup>JTAG speed

--macro

Syntax	<code>--macro filename</code>	
Parameters	<i>filename</i>	The C-SPY macro file to be used (filename extension <code>mac</code>).
Applicability	Sent to <code>cspybat</code> .	
Description	Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.	
See also	<i>Briefly about using C-SPY macros</i> , page 268.	

--mac_handler_address

Syntax	<code>--mac_handler_address=address</code>	
Parameters	<i>address</i>	The start address of the memory area for the debug handler.
Applicability	The C-SPY Macraigor driver.	
Description	Use this option to specify the location—the memory address—of the debug handler used by Intel XScale devices.	

See also *Debug handler address*, page 377.



Project>Options>Debugger>Macraigor>Debug handler address

--mac_interface

Syntax	<code>--mac_interface={JTAG SWD}</code>	
Parameters	JTAG	Uses JTAG communication with the target system (default).
	SWD	Uses SWD communication with the target system (Cortex-M only); uses fewer pins than JTAG communication.
Applicability	The C-SPY Macraigor driver.	
Description	Use this option to specify the communication channel between the Macraigor debug probe and the target system.	



Project>Options>Debugger>Macraigor>Interface

--mac_jtag_device

Syntax	<code>--mac_jtag_device=device</code>	
Parameters	<i>device</i>	The device corresponding to the hardware interface that is used. Choose between Macraigor <code>mpDemon</code> , <code>usbdemon</code> , and <code>usb2demon</code> .
Applicability	The C-SPY Macraigor driver.	
Description	Use this option to select the device corresponding to the hardware interface that is used.	
See also	<i>OCD interface device</i> , page 375.	



Project>Options>Debugger>Macraigor>OCD interface device

--mac_multiple_targets

Syntax	<code>--mac_multiple_targets=<tap-no>@dev0, dev1, dev2, dev3, . . .</code>	
Parameters	<code>tap-no</code>	The TAP number of the device to connect to, where 0 connects to the first device, 1 to the second, and so on.
	<code>dev0-devn</code>	The nearest TDO pin on the Macraigor JTAG probe.
Applicability	The C-SPY Macraigor driver.	
Description	If there is more than one device on the JTAG scan chain, each device must be defined. Use this option to specify which device you want to connect to.	
Example	<code>--mac_multiple_targets=0@ARM7TDMI, ARM7TDMI</code>	
See also	<i>JTAG scan chain with multiple targets</i> , page 376.	



Project>Options>Debugger>Macraigor>JTAG scan chain with multiple targets

--mac_reset_pulls_reset


Syntax	<code>--mac_reset_pulls_reset=time</code>	
Parameters	<code>time</code>	0–2000 which is the delay in milliseconds after reset.
Applicability	The C-SPY Macraigor driver.	
Description	Use this option to make C-SPY perform an initial hardware reset when the debugger is started, and to specify the delay for the reset.	
See also	<i>Hardware reset</i> , page 376.	



Project>Options>Debugger>Macraigor>Hardware reset

--mac_set_temp_reg_buffer

Syntax	<code>--mac_set_temp_reg_buffer=address</code>
--------	--

Parameters	<i>address</i>	The start address of the RAM area.
Applicability	The C-SPY Macraigor driver.	
Description	Use this option to specify the start address of the RAM area that is used for controlling the MMU and caching via the CP15 coprocessor.	
		To set this option, use Project>Options>Debugger>Extra Options .

--mac_speed

Syntax	<code>--mac_speed={ <i>factor</i> }</code>	
Parameters	<i>factor</i>	The factor by which the JTAG probe clock is divided when generating the scan clock. The number must be in the range 1–8 where 1 is the fastest.
Applicability	The C-SPY Macraigor driver.	
Description	Use this option to set the JTAG speed between the JTAG probe and the ARM JTAG ICE port.	
See also	<i>JTAG speed</i> , page 375.	



Project>Options>Debugger>Macraigor>JTAG speed

--mac_xscale_ir7

Syntax	<code>--mac_xscale_ir7</code>	
Applicability	The C-SPY Macraigor driver.	
Description	Use this option to specify that the XScale ir7 core is used, instead of XScale ir5. Note that this option is mandatory when using the XScale ir7 core. These XScale cores are supported by the C-SPY Macraigor driver: Intel XScale Core 1 (5-bit instruction register—ir5) Intel XScale Core 2 (7-bit instruction register—ir7)	



To set this option, use **Project>Options>Debugger>Extra Options**.

--mapu

Syntax	--mapu
Applicability	The C-SPY simulator driver.
Description	Specify this option to use the section information in the debug file for memory access checking. During the execution, the simulator will then check for accesses to unspecified memory ranges. If any such access is found, the C function call stack and a message will be printed on <code>stderr</code> and the execution will stop.
See also	<i>Memory access checking</i> , page 142.



To set related options, choose:
Simulator>Memory Access Setup

-p

Syntax	-p <i>filename</i>
Parameters	<i>filename</i> 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	<i>Selecting a device description file</i> , page 47.

--plugin

Syntax	--plugin <i>filename</i>
Parameters	<i>filename</i> The plugin file to be used (filename extension <code>dll</code>).
Applicability	Sent to <code>cspybat</code> .

Description	<p>Certain C/C++ standard library functions, for example <code>printf</code>, 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 <code>cspybat</code>, a dedicated plugin module called <code>armbat.dll</code> located in the <code>arm\bin</code> directory must be used.</p> <p>Use this option to include this plugin during the debug session. This option can be used more than once on the command line.</p> <p>Note: You can use this option to include also other plugin modules, but in that case the module must be able to work with <code>cspybat</code> specifically. This means that the C-SPY plugin modules located in the <code>common\plugin</code> directory cannot normally be used with <code>cspybat</code>.</p>
-------------	---

--proc_stack_stack

Syntax	<pre>--proc_stack_stack=startaddress,endaddress</pre> <p>where <i>stack</i> is one of <code>main</code> or <code>proc</code> for Cortex-M and</p> <p>where <i>stack</i> is one of <code>usr</code>, <code>svc</code>, <code>irq</code>, <code>fiq</code>, <code>und</code>, or <code>abt</code> for other ARM cores</p>				
Parameters	<table><tr><td><i>startaddress</i></td><td>The start address of the stack, specified either as a value or as an expression.</td></tr><tr><td><i>endaddress</i></td><td>The end address of the stack, specified either as a value or as an expression.</td></tr></table>	<i>startaddress</i>	The start address of the stack, specified either as a value or as an expression.	<i>endaddress</i>	The end address of the stack, specified either as a value or as an expression.
<i>startaddress</i>	The start address of the stack, specified either as a value or as an expression.				
<i>endaddress</i>	The end address of the stack, specified either as a value or as an expression.				
Applicability	All C-SPY drivers. Note that this command line option is only available when using C-SPY from the IDE; not in batch mode using <code>cspybat</code> .				
Description	Use this option to provide information to the C-SPY stack plugin module about reserved stacks. By default, C-SPY receives this information from the system startup code, but if you for some reason want to override the default values, this option can be useful.				
Example	<pre>--proc_stack_irq=0x8000,0x80FF</pre>				



To set this option, use **Project>Options>Debugger>Extra Options**.

--rdi_allow_hardware_reset

Syntax	<pre>--rdi_allow_hardware_reset</pre>
Applicability	The C-SPY RDI driver.

Description	Use this option to allow the emulator to perform a hardware reset of the target. Requires support by the emulator.
See also	<i>Allow hardware reset</i> , page 377.



Project>Options>Debugger>RDI>Allow hardware reset

--rdi_driver_dll

Syntax	<code>--rdi_driver_dll <i>filename</i></code>	
Parameters	<i>filename</i>	The file or path to the driver DLL file.
Applicability	The C-SPY RDI driver The C-SPY JTAGjet driver.	
Description	Use this option to specify the path to the driver DLL file provided with the JTAG pod.	
See also	<i>Manufacturer RDI driver</i> , page 377.	



Project>Options>Debugger>RDI>Manufacturer RDI driver

For the JTAGjet driver, this option is not available in the IDE.

--rdi_step_max_one

Syntax	<code>--rdi_step_max_one</code>	
Applicability	The C-SPY Angel debug monitor driver The C-SPY RDI driver.	
Description	Use this option to execute only one instruction. The debugger will turn off interrupts while stepping and, if necessary, simulate the instruction instead of executing it.	



To set this option, use **Project>Options>Debugger>Extra Options**.

--semihosting

Syntax	<code>--semihosting={none iar_breakpoint}</code>	
Parameters	No parameter	Use standard semihosting.
	none	Does not use semihosted I/O.
	iar_breakpoint	Uses the IAR proprietary semihosting variant.
Applicability	All C-SPY drivers.	
Description	<p>Use this option to enable semihosted I/O and to choose the kind of semihosting interface to use. Note that if this option is not used, semihosting will by default be enabled and C-SPY will try to choose the correct semihosting mode automatically. This means that normally you do not have to use this option if your application is linked with semihosting.</p> <p>To make semihosting work, your application must be linked with a semihosting library.</p>	
See also	The <i>IAR C/C++ Development Guide for ARM</i> for more information about linking with semihosting.	



Project>Options>General Options>Library Configuration

--silent

Syntax	<code>--silent</code>
Applicability	Sent to cspybat.
Description	Use this option to omit the sign-on message.

--stlink_interface

Syntax	<code>--stlink_interface={JTAG SWD}</code>	
Parameters	JTAG	Uses JTAG communication with the target system (default).
	SWD	Uses SWD communication with the target system.

Applicability	The C-SPY ST-LINK driver.
Description	Use this option to specify the communication channel between the ST-LINK debug probe and the target system.
See also	<i>Interface</i> , page 379.



Project>Options>Debugger>ST-LINK>ST-LINK>Interface

--stlink_reset_strategy

Syntax	<code>--stlink_reset_strategy=delay, strategy</code>	
Parameters	<i>delay</i>	The delay time measured in milliseconds. <i>delay</i> is ignored and should be 0.
	<i>strategy</i>	The reset strategy. 0, (Normal) performs the standard reset procedure. 1, (Reset Pin) uses the reset pin to perform a hardware reset. Only available for ST-LINK version 2. 2, (Connect during reset) ST-LINK connects to the target while keeping Reset active (Reset is pulled low and remains low while connecting to the target). Only available for ST-LINK version 2.


Applicability	The C-SPY ST-LINK driver.
Description	Use this option to select the reset strategy to be used at debugger startup.
See also	<i>Reset</i> , page 379




Project>Options>Debugger>ST-LINK>Setup>Reset

--timeout

Syntax	<code>--timeout milliseconds</code>	
Parameters	<i>milliseconds</i>	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.

--xds_rootdir

Syntax	<code>--xds_rootdir=path</code>
Applicability	The C-SPY TI XDS100 driver
Description	Use this option to specify the path to the directory where the TI XDS100 driver package is installed.
	 To set this option, use Project>Options>Debugger>Extra Options .

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 reference information on the generic options, see:

- *Setup*, page 355
- *Download*, page 357
- *Extra Options*, page 358
- *Images*, page 359
- *Plugins*, page 360.

- 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 Angel debug monitor driver	<i>Angel</i> , page 361
C-SPY GDB Server driver	<i>GDB Server</i> , page 363 <i>Breakpoints options</i> , page 132

Table 44: Options specific to the C-SPY drivers you are using

C-SPY driver	Available options pages
C-SPY IAR ROM-monitor driver	IAR ROM-monitor, page 364
C-SPY J-Link/J-Trace driver	Setup options for J-Link/J-Trace, page 365 Connection options for J-Link/J-Trace, page 369 Breakpoints options, page 132
C-SPY TI Stellaris driver	Setup options for TI Stellaris, page 380
C-SPY TI XDS100 driver	Setup options for TI XDS100, page 381
C-SPY Macraigor driver	Macraigor, page 375
C-SPY JTAGjet driver	JTAGjet, page 371 RDI Configuration dialog box for JTAGjet, page 373
RDI driver	RDI, page 377
ST-LINK driver	ST-LINK, page 379
Third-party driver	Third-Party Driver options, page 382.

Table 44: 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.

The Setup dialog box contains the following elements:

- Driver:** A dropdown menu currently showing 'Simulator'.
- Run to:** A checked checkbox followed by a text field containing 'main'.
- Setup macros:** A section containing a checked 'Use macro file(s)' checkbox and two empty text input fields, each with a browse button (three dots).
- Device description file:** A section containing an unchecked 'Override default' checkbox and one empty text input field with a browse button.

Figure 118: Debugger setup options

Driver

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

Simulator

Angel

GDB Server

IAR ROM-monitor

J-Link/J-Trace

TI Stellaris

TI XDS100

Macraigor

JTAGjet

RDI

ST-LINK

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.

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.

It is possible to specify up to two different macro files.

Device description file

A default device description file—either an IAR-specific `ddf` file or a CMSIS System View 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 51.

IAR-specific device description files for each ARM device are provided in the directory `arm\config` and have the filename extension `ddf`.

Download

By default, C-SPY downloads the application to RAM or flash when a debug session starts. The **Download** options let you modify the behavior of the download.

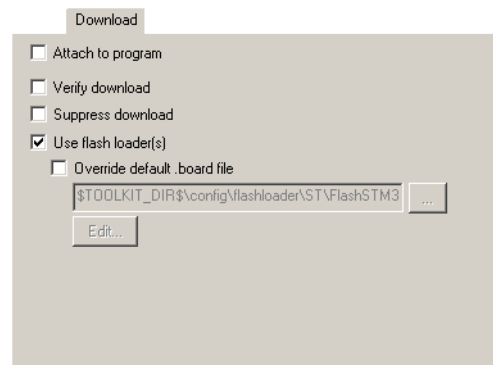


Figure 119: C-SPY Download options

Attach to program

Makes the debugger attach to a running application at its current location, without resetting or halting (for J-Link only) the target system. To avoid unexpected behavior when using this option, the **Debugger>Setup** option **Run to** should be deselected.

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

Suppress download

Disables the downloading of code, while preserving the present content of the flash. This command is useful if you want to debug an application that already resides in target memory.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Note: It is important that the image that resides in target memory is linked consistently with how you use C-SPY for debugging. This applies, for example, if you first link your application using an output format without debug information, such as Intel-hex, and then load the application separately from C-SPY. If you then use C-SPY only for debugging without downloading, you cannot build the debugged application with any of the options **Semihosted** or **IAR breakpoint**—on the **General Options>Library**

Configuration page—as that would add extra code, resulting in two different code images.

Use flash loader(s)

Use this option to use one or several flash loaders for downloading your application to flash memory. If a flash loader is available for the selected chip, it is used by default. Press the **Edit** button to display the **Flash Loader Overview** dialog box.

For more information about flash loaders, see *Using flash loaders*, page 397.

Override default .board file

A default flash loader is selected based on your choice of device on the **General Options>Target** page. To override the default flash loader, select **Override default .board file** and specify the path to the flash loader you want to use. A browse button is available for your convenience. Click **Edit** to display the **Flash Loader Overview** dialog box. For more information, see *Flash Loader Overview dialog box*, page 399.

Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.

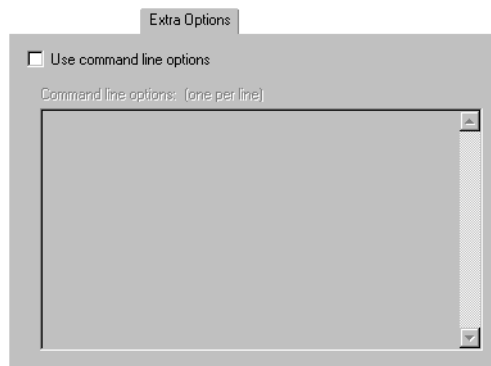


Figure 120: Debugger extra options

Use command line options

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

Images

The **Images** options control the use of additional debug files to be downloaded.

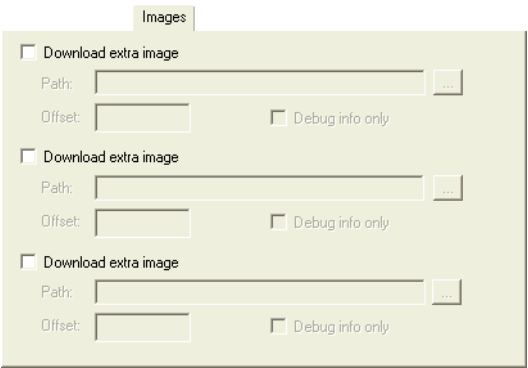


Figure 121: Debugger images options

Download extra Images

Controls the use of additional debug files to be downloaded:

- | | |
|------------------------|---|
| Path | Specify the debug file to be downloaded. A browse button is available for your convenience. |
| Offset | Specify an integer that determines the destination address for the downloaded debug file. |
| Debug info only | Makes the debugger download only debug information, and not the complete debug file. |

If you want to download more than three images, use the related C-SPY macro, see [__loadImage](#), page 297.

For more information, see *Loading multiple images*, page 50.

Plugins

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

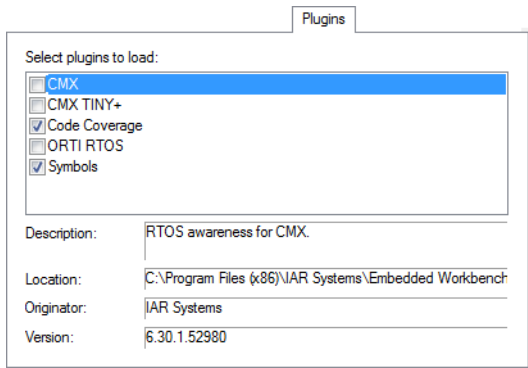


Figure 122: Debugger plugin options

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 `arm\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.

More specifically, you will get information about:

- *Angel*, page 361
- *GDB Server*, page 363
- *IAR ROM-monitor*, page 364
- *Setup options for J-Link/J-Trace*, page 365
- *Connection options for J-Link/J-Trace*, page 369
- *JTAGjet*, page 371
- *RDI Configuration dialog box for JTAGjet*, page 373
- *Macraigor*, page 375
- *RDI*, page 377
- *ST-LINK*, page 379
- *Setup options for TI Stellaris*, page 380
- *Setup options for TI XDS100*, page 381
- *Third-Party Driver options*, page 382.

Angel

The **Angel** options control the C-SPY Angel debug monitor driver.

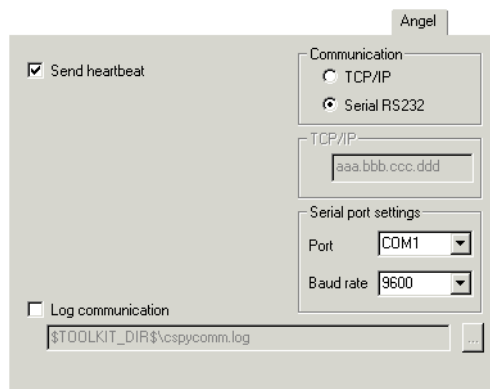


Figure 123: C-SPY Angel options

Send heartbeat

Makes C-SPY poll the target system periodically while your application is running. That way, the debugger can detect if the target application is still running or has terminated abnormally. Enabling the heartbeat will consume some extra CPU cycles from the running program.

Communication

Selects the Angel communication link. RS232 serial port connection and TCP/IP via an Ethernet connection are supported.

TCP/IP

Specify the IP address of the target device in the text box.

Serial port settings

Configures the serial port. You can specify

- | | |
|------------------|---|
| Port | Selects which port on the host computer to use as the Angel communication link. |
| Baud rate | Sets the communication speed. |

The initial Angel serial speed is always 9600 baud. After the initial handshake, the link speed is changed to the specified speed. Communication problems can occur at very high speeds; some Angel-based evaluation boards will not work above 38,400 baud.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the Angel monitor protocol is required.

GDB Server

The **GDB Server** options control the C-SPY GDB Server for the STR9-comStick evaluation board.



Figure 124: GDB Server options

TCP/IP address or hostname

Specify the IP address and port number of a GDB server; by default the port number 3333 is used. The TCP/IP connection is used for connecting to a J-Link server running on a remote computer.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the JTAG interface is required.

IAR ROM-monitor

The **IAR ROM-monitor** options control the C-SPY IAR ROM-monitor interface.

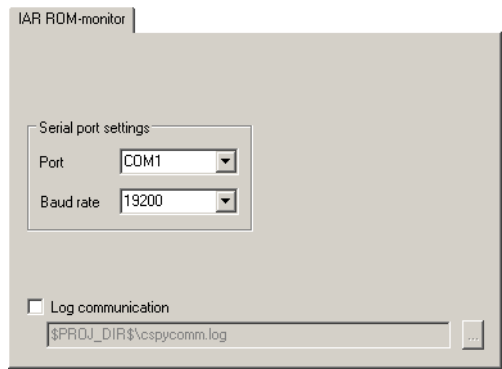


Figure 125: IAR ROM-monitor options

Serial port settings

Configures the serial port. You can specify

- | | |
|------------------|---|
| Port | Selects which port on the host computer to use as the ROM-monitor communication link. |
| Baud rate | Sets the communication speed. The serial port communication link speed must match the speed selected on the target board. |

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the ROM-monitor protocol is required.

Setup options for J-Link/J-Trace

The **Setup** options specify the J-Link/J-Trace probe.

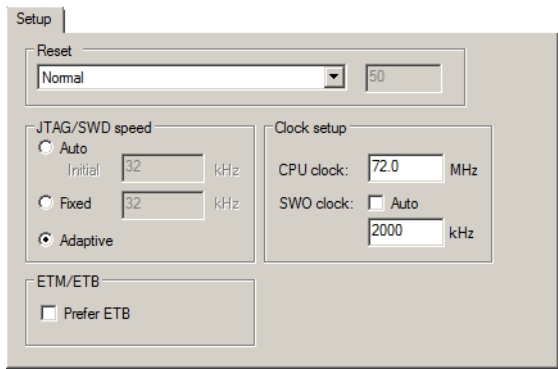


Figure 126: J-Link/J-Trace Setup options

Reset

Selects the reset strategy to be used when the debugger starts. Note that Cortex-M uses a different set of strategies than other devices. The actual reset strategy type number is specified for each available choice. Choose between:

- Normal** (0, default) Tries to reset the core via the reset strategy Core and peripherals first. If this fails, the reset strategy Core only is used. It is recommended that you use this strategy to reset the target.
- Core** (1) Resets the core via the VECTRESET bit; the peripheral units are not affected.
- Core and peripherals** (8) Resets the core and the peripherals.
- Reset Pin** (2) J-Link pulls its RESET pin low to reset the core and the peripheral units. Normally, this causes the CPU RESET pin of the target device to go low as well, which results in a reset of both the CPU and the peripheral units.
- Connect during reset** (3) J-Link connects to the target while keeping Reset active (reset is pulled low and remains low while connecting to the target). This is the recommended reset strategy for STM32 devices. This strategy is available for STM32 devices only.

Halt after bootloader (4 or 7)	<p>NXP Cortex-M0 devices. This is the same strategy as the Normal strategy, but the target is halted when the bootloader has finished executing. This is the recommended reset strategy for LPC11xx and LPC13xx devices.</p> <p>Analog Devices Cortex-M3 devices (7), Resets the core and peripheral units by setting the <code>SYSRESETREQ</code> bit in the <code>AIRCR</code>. The core is allowed to perform the ADI kernel (which enables the debug interface), but the core is halted before the first instruction after the kernel is executed to guarantee that no user application code is performed after reset.</p>
Halt before bootloader (5)	<p>This is the same strategy as the Normal strategy, but the target is halted before the bootloader has started executing. This strategy is normally not used, except in situations where the bootloader needs to be debugged. This strategy is available for LPC11xx and LPC13xx devices only.</p>
Normal, disable watchdog (6, 9, or 10)	<p>First performs a Normal reset, to reset the core and peripheral units and halt the CPU immediately after reset. After the CPU is halted, the watchdog is disabled, because the watchdog is by default running after reset. If the target application does not feed the watchdog, J-Link loses connection to the device because it is permanently reset. This strategy is available for Freescale Kinetis devices (6), for NXP LPC 1200 devices (9), and for Samsung S3FN60D devices (10).</p>

All of these strategies are available for both the JTAG and the SWD interface, and all strategies halt the CPU after the reset.

For other cores, choose between these strategies:

Hardware, halt after delay (ms) (0)	<p>Specify the delay between the hardware reset and the halt of the processor. This is used for making sure that the chip is in a fully operational state when C-SPY starts to access it. By default, the delay is set to zero to halt the processor as quickly as possible.</p> <p>This is a hardware reset.</p>
--	---

Hardware, halt using Breakpoint (1)	<p>After reset, J-Link continuously tries to halt the CPU using a breakpoint. Typically, this halts the CPU shortly after reset; the CPU can in most systems execute some instructions before it is halted.</p> <p>This is a hardware reset.</p>
Hardware, halt at 0 (4)	<p>Halts the processor by placing a breakpoint at the address zero. Note that this is not supported by all ARM microcontrollers.</p> <p>This is a hardware reset.</p>
Hardware, halt using DBGRQ (5)	<p>After reset, J-Link continuously tries to halt the CPU using DBGRQ. Typically, this halts the CPU shortly after reset; the CPU can in most systems execute some instructions before it is halted.</p> <p>This is a hardware reset.</p>
Software (-)	<p>Sets PC to the program entry address.</p> <p>This is a software reset.</p>
Software, Analog devices (2)	<p>Uses a reset sequence specific for the Analog Devices ADuC7xxx family. This strategy is only available if you have selected such a device from the Device drop-down list on the General Options>Target page.</p> <p>This is a software reset.</p>
Hardware, NXP LPC (9)	<p>This strategy is only available if you have selected such a device from the Device drop-down list on the General Options>Target page.</p> <p>This is a hardware reset specific to NXP LPC devices.</p>
Hardware, Atmel AT91SAM7 (8)	<p>This strategy is only available if you have selected such a device from the Device drop-down list on the General Options>Target page.</p> <p>This is a hardware reset specific for the Atmel AT91SAM7 family.</p>

For more details about the different reset strategies, see the *IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores* available in the `arm\doc` directory.

A software reset of the target does not change the settings of the target system; it only resets the program counter and the mode register CPSR to its reset state. Normally, a

C-SPY reset is a software reset only. If you use the **Hardware reset** option, C-SPY will generate an initial hardware reset when the debugger is started. This is performed once before download, and if the option **Use flash loader(s)** is selected, also once after flash download, see Figure 5, *Debugger startup when debugging code in flash*, and Figure 6, *Debugger startup when debugging code in RAM*.



Hardware resets can be a problem if the low-level setup of your application is not complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 52.

JTAG/SWD speed

Specify the JTAG communication speed in kHz. Choose between:

Auto	<p>Automatically uses the highest possible frequency for reliable operation. The initial speed is the fixed frequency used until the highest possible frequency is found. The default initial frequency—32 kHz—can normally be used, but in cases where it is necessary to halt the CPU after the initial reset, in as short time as possible, the initial frequency should be increased.</p> <p>A high initial speed is necessary, for example, when the CPU starts to execute unwanted instructions—for example power down instructions—from flash or RAM after a reset. A high initial speed would in such cases ensure that the debugger can quickly halt the CPU after the reset.</p> <p>The initial value must be in the range 1–12000 kHz.</p>
Fixed	<p>Sets the JTAG communication speed in kHz. The value must be in the range 1–12000 kHz.</p> <p>If there are JTAG communication problems or problems in writing to target memory (for example during program download), these problems might be avoided if the speed is set to a lower frequency.</p>
Adaptive	<p>Works only with ARM devices that have the <code>RTCK</code> JTAG signal available. For more information about adaptive speed, see the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> available in the <code>arm\doc</code> directory.</p>

Clock setup

Specifies the CPU clock. Choose between:

- CPU clock

Specifies the exact clock frequency used by the internal processor clock, `HCLK`, in MHz. The value can have decimals. This value is used for configuring the SWO communication speed and for calculating timestamps.
- SWO clock

Specifies the clock frequency of the SWO communication channel in KHz.
- Auto

Automatically uses the highest possible frequency that the J-Link debug probe can handle. If Auto is not selected, the wanted SWO clock value can be input in the text box. The value can have decimals. Use this option if data packets are lost during transmission.

To override the **Clock setup** options, use the **Override project default** option in the **SWO Configuration** dialog box, see *Override project default*, page 182.

ETM/ETB

The **Prefer ETB** option selects ETB trace instead of ETM trace, which is the default.
Note: This option applies only to J-Trace.

Connection options for J-Link/J-Trace

The **Connection** options specify the connection with the J-Link/J-Trace probe.

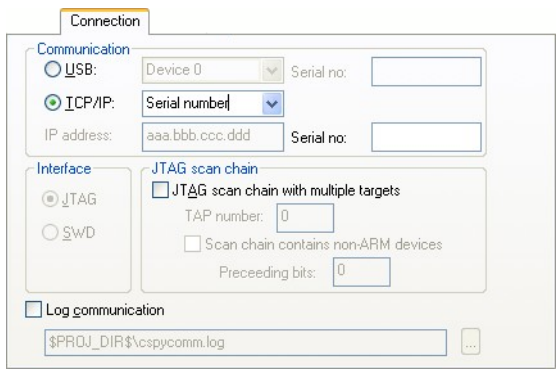


Figure 127: J-Link/J-Trace Connection options

Communication

Selects the communication channel between C-SPY and the J-Link debug probe.
Choose between:

- USB

Selects the USB connection. If Serial number is selected in the drop-down list, the J-Link debug probe with the specified serial number is chosen.
- TCP/IP

Specify the IP address of a J-Link server. The TCP/IP connection is used for connecting to a J-Link server running on a remote computer.

IP address, specify the IP address of a J-Link probe connected to LAN.

Auto detect, automatically scans the network for J-Link probes. Use the dialog box to choose among the detected J-Link probes.

Serial number, connects to the J-Link probe on the network with the serial number that you specify.

Interface

Selects the communication interface between the J-Link debug probe and the target system. Choose between:

- JTAG (default)

Uses the JTAG interface.
- SWD

Uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the **General Options>Library Configuration** page, SWD is selected automatically. For more information about SWO settings, see *SWO Trace Window Settings dialog box*, page 179.

JTAG scan chain

Specifies the JTAG scan chain. Choose between:

- JTAG scan chain with multiple targets** Specifies that there is more than one device on the JTAG scan chain.
- TAP number** Specify the TAP (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero.
- Scan chain contains non-ARM devices** Enables JTAG scan chains that mix ARM devices with other devices like, for example, FPGA.
- Preceding bits** Specify the number of IR bits before the ARM device to be debugged.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the JTAG interface is required.

JTAGjet

The JTAGjet options specify the JTAGjet interface.

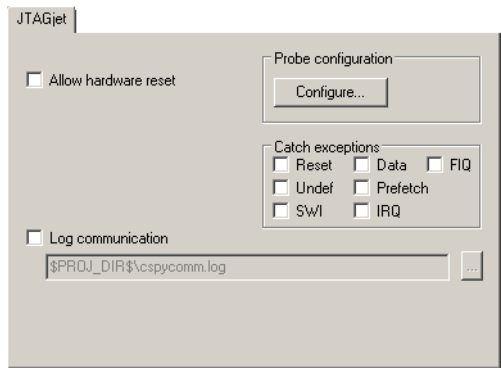


Figure 128: JTAGjet options

Allow hardware reset

Allows the emulator to perform a hardware reset of the target.

A software reset of the target does not change the settings of the target system; it only resets the program counter to its reset state.



You should only allow hardware resets if the low-level setup of your application is complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 52.

Probe configuration

Click the **Configure** button to display the **RDI Configuration** dialog box, see *RDI Configuration dialog box for JTAGjet*, page 373.

Catch exceptions

Causes exceptions to be treated as breakpoints. Instead of handling the exception as defined by the running program, the debugger will stop when the exception occurs.
The ARM core exceptions that can be caught are:

Exception	Description
Reset	Reset
Undef	Undefined instruction
SWI	Software interrupt
Data	Data abort (data access memory fault)
Prefetch	Prefetch abort (instruction fetch memory fault)
IRQ	Normal interrupt
FIQ	Fast interrupt

Table 45: Catching exceptions

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.

RDI Configuration dialog box for JTAGjet

The **RDI Configuration** dialog box is available by choosing **Project>Options>JTAGjet>Configure**.

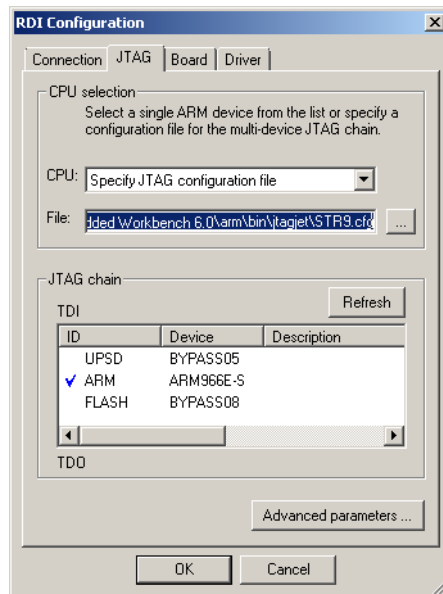


Figure 129: The RDI Configuration dialog box

To configure the JTAGjet connection:

- 1 Make sure that the **Connection** tab is selected.
 - To establish a debugger-emulator connection with minimum effort, click the **Connect Automatically** button. The connection parameters appear in the **Status** box.
 - For additional control over the connection, click the **Specify Connection** button to set up the communication port and its parameters manually.
- 2 Once the connection has been established, select the **JTAG** tab. Choose your target device from the **CPU** drop-down list as follows.
 - If your JTAG chain contains a single ARM device, select the device from the **CPU** drop-down list. The name (ID) and device type, along with a short description, are displayed in the **JTAG Chain** group box.
 - Use a JTAG configuration file to specify your JTAG chain if your JTAG chain contains multiple devices. From the **CPU** drop-down list, select **Specify JTAG**

Configuration File. Enter the file path and name in the **File** text box. Alternatively, use the file browser button located next to the **File** text box to navigate to your file. After the file has been loaded, the chain description is displayed in the **JTAG Chain** group box. In the **ID** column, select the device you want to debug. Note that you cannot select a bypass device.

- Click the **Advanced Parameters** button. Select **ARM** from the **Emulation Mode** drop-down list for all but TI ARM925-based devices. For the TI ARM925-based devices, choose TI Emulation Mode. Leaving the **JTAG Clock** and **JTAG Header** fields empty lets the emulator select those parameters automatically, depending on the device type. However, when using the JTAG cable labeled TI-14 (formerly TMS320), TI-14-ISO (formerly ISO-TMS320), or CTI-20, select the **TI JTAG Header** option. Otherwise, the emulator might incorrectly report that the device is held in reset. For the XScale device, set the **JTAG Header** field to the following: ARM,TRST:1.
- 3** Click the **Board** tab and select the byte order of your target board. In the **File** field, enter the name of the startup macro file. If you are using a popular evaluation board, chances are that an appropriate macro has been provided for you. Otherwise, you might need to write your own or leave the field blank. The **Debug Handler Address** field and the **Vector Changes in Runtime** field apply to the XScale processor only. It might be necessary to increase the **Sleep After CPU Reset** value if the XScale processor cannot be halted. Select the **Flash Write Enable** option to load the code to flash memory using the flash programmer built into the JTAGjet driver.
 - 4** Click the **Driver** tab to configure driver protocol logs and error reports.
 - To display the driver protocol log in the Log window, select **Log Enable**. If you also need to store the log in a file, enter the filename in the **File** text box.
 - To enable the driver to generate descriptive error messages, select **Show Error Messages**. This option does not affect the way your debugger displays its own error messages; it is designed simply to augment and clarify those debugger messages that tend to be cryptic or are limited to error codes only.
 - 5** Click **OK**. In the **Choose Target** dialog box, click **OK** again to accept your settings and connect to the target device.

Macraigor

The **Macraigor** options specify the Macraigor interface.

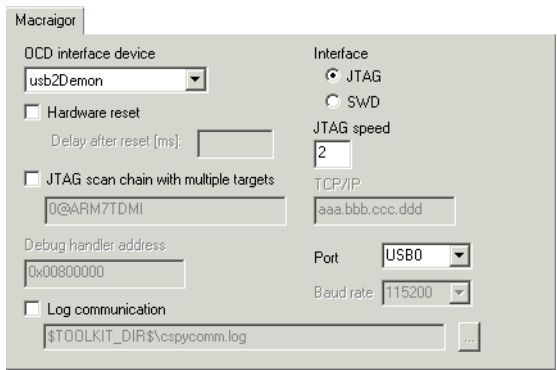
The image shows a software configuration window titled "Macraigor". It contains several settings: "OCD interface device" is a dropdown menu with "usb2Demon" selected; "Interface" has two radio buttons, "JTAG" (selected) and "SWD"; "JTAG speed" is a numeric input field with the value "2"; "TCP/IP" is a text field with "aaa.bbb.ccc.ddd"; "Port" is a dropdown menu with "USB0" selected; "Baud rate" is a dropdown menu with "115200" selected; "Hardware reset" is an unchecked checkbox; "Delay after reset [ms]" is an empty numeric input field; "JTAG scan chain with multiple targets" is an unchecked checkbox; "Debug handler address" is a text field with "0x00800000"; "Log communication" is an unchecked checkbox; and a file path "\$TOOLKIT_DIR\$\cspycomm.log" is shown in a text field with a browse button (three dots) to its right.

Figure 130: Macraigor options

OCD interface device

Selects the device corresponding to the hardware interface you are using. Supported Macraigor JTAG probes is Macraigor **mpDemon**.

Interface

Selects the communication interface between the J-Link debug probe and the target system. Choose between:

- | | |
|-----------------------|---|
| JTAG (default) | Uses the JTAG interface. |
| SWD | Uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the General Options>Library Configuration page, SWD is selected automatically. For more information about SWO settings, see <i>SWO Trace Window Settings dialog box</i> , page 179. |

JTAG speed

Specify the speed between the JTAG probe and the ARM JTAG ICE port. The number must be in the range 1–8 and sets the factor by which the JTAG probe clock is divided when generating the scan clock.



The mpDemon interface might require a higher setting such as 2 or 3, that is, a lower speed.

TCP/IP

Specify the IP address of a JTAG probe connected to the Ethernet/LAN port.

Port

Selects which serial port or parallel port on the host computer to use as communication link. Select the host port to which the JTAG probe is connected.

In the case of parallel ports, you should normally use LPT1 if the computer is equipped with a single parallel port. Note that a laptop computer might in some cases map its single parallel port to LPT2 or LPT3. If possible, configure the parallel port in EPP mode because this mode is fastest; bidirectional and compatible modes will work but are slower.

Baud rate

Selects the serial communication speed.

Hardware reset

Generates an initial hardware reset when the debugger is started. This is performed once before download, and if the option **Use flash loader(s)** is selected, also once after flash download, see *Debugger startup when debugging code in flash*, page 53, and Figure 6, *Debugger startup when debugging code in RAM*.

A software reset of the target does not change the settings of the target system; it only resets the program counter to its reset state. Normally, a C-SPY reset is a software reset only.



Hardware resets can be a problem if the low-level setup of your application is not complete. If low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 52.

JTAG scan chain with multiple targets

Defines each device on the JTAG scan chain, if there is more than one. Also, you must state which device you want to connect to. The syntax is:

```
<0>@dev0, dev1, dev2, dev3, . . .
```

where 0 is the TAP number of the device to connect to, and `dev0` is the nearest TDO pin on the Macraigor JTAG probe.

Debug handler address

Specify the location—the memory address—of the debug handler used by Intel XScale devices. To save memory space, you should specify an address where a small portion of cache RAM can be mapped, which means the location should not contain any physical memory. Preferably, find an unused area in the lower 16-Mbyte memory and place the handler address there.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the JTAG interface is required.

RDI

With the **RDI** options you can use JTAG interfaces compliant with the ARM Ltd. RDI 1.5.1 specification. One example of such an interface is the ARM RealView Multi-ICE JTAG interface.

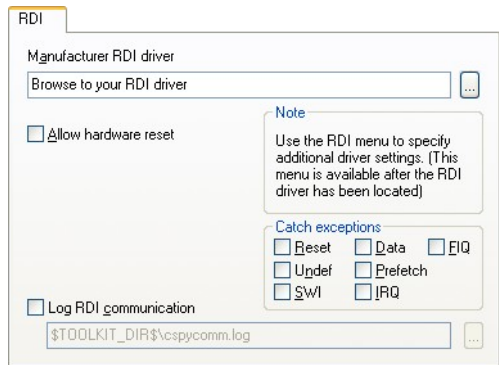


Figure 131: RDI options

Manufacturer RDI driver

Specify the file path to the RDI driver DLL file provided with the JTAG pod.

Allow hardware reset

Allows the emulator to perform a hardware reset of the target.

A software reset of the target does not change the settings of the target system; it only resets the program counter to its reset state.



You should only allow hardware resets if the low-level setup of your application is complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 52.

Note: This option requires that hardware resets are supported by the RDI driver you are using.

Catch exceptions

Causes exceptions to be treated as breakpoints. Instead of handling the exception as defined by the running program, the debugger will stop.

The ARM core exceptions that can be caught are:

Exception	Description
Reset	Reset
Undef	Undefined instruction
SWI	Software interrupt
Data	Data abort (data access memory fault)
Prefetch	Prefetch abort (instruction fetch memory fault)
IRQ	Normal interrupt
FIQ	Fast interrupt

Table 46: Catching exceptions

Log RDI communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the RDI interface is required.

ST-LINK

The **ST-LINK** page contains options for the ST-LINK probe.

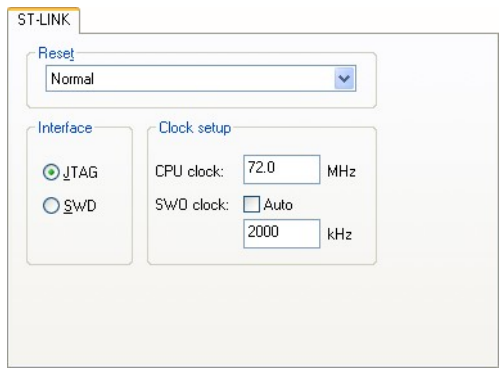


Figure 132: ST-LINK Setup options

Reset

Selects the reset strategy to be used when the debugger starts. The actual reset strategy type number is specified for each available choice. Choose between:

- Normal** (0)
- Performs the standard reset procedure.
- Reset Pin** (1)
- Uses the reset pin to perform a hardware reset. Only available for ST-LINK version 2.
- Connect during reset** (2)
- ST-LINK connects to the target while keeping the reset pin active (the reset pin is pulled low and remains low while connecting to the target). Only available for ST-LINK version 2.

Interface

Selects the communication interface between the ST-LINK debug probe and the target system. Choose between:

- JTAG** (default)
- Uses the JTAG interface.
- SWD**
- Uses fewer pins than JTAG.

Clock setup

Specifies the CPU clock. Choose between:

- CPU clock

Specifies the exact clock frequency used by the internal processor clock, `HCLK`, in MHz. The value can have decimals. This value is used for configuring the SWO communication speed and for calculating timestamps.
- SWO clock

Specifies the clock frequency of the SWO communication channel in KHz.
- Auto

Automatically uses the highest possible frequency that the J-Link debug probe can handle. If Auto is not selected, the wanted SWO clock value can be input in the text box. The value can have decimals. Use this option if data packets are lost during transmission.

To override the **Clock setup** options, use the **Override project default** option in the **SWO Configuration** dialog box, see *Override project default*, page 182.

Setup options for TI Stellaris

The **Setup** options specify the TI Stellaris interface.

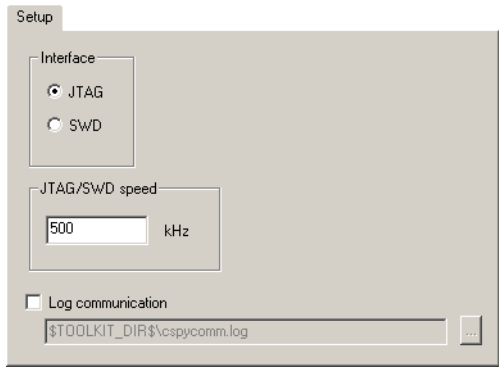


Figure 133: TI Stellaris Setup options

Interface

Selects the communication interface between the J-Link debug probe and the target system. Choose between:

JTAG (default)	Uses the JTAG interface.
SWD	Uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the General Options>Library Configuration page, SWD is selected automatically. For more information about SWO settings, see <i>SWO Trace Window Settings dialog box</i> , page 179.

JTAG/SWD speed

Specify the JTAG communication speed in kHz.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge about the communication protocol is required.

Setup options for TI XDS100

There are no options on this page.

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.

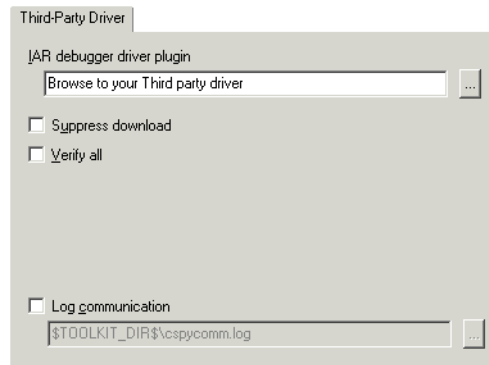


Figure 134: C-SPY Third-Party Driver options

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 it 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
- The C-SPY GDB Server driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY JTAGjet driver
- The C-SPY Macraigor driver
- The C-SPY RDI driver
- The C-SPY ST-LINK driver
- The C-SPY TI Stellaris driver
- The C-SPY TI XDS100 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 386

Simulator menu

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

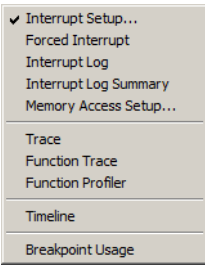


Figure 135: Simulator menu

These commands are available on the menu:

Interrupt Setup	Displays a dialog box where you can configure C-SPY interrupt simulation, see <i>Interrupt Setup dialog box</i> , page 254.
Forced Interrupts	Opens a window from where you can instantly trigger an interrupt, see <i>Forced Interrupt window</i> , page 257.
Interrupt Log	Opens a window which displays the status of all defined interrupts, see <i>Interrupt Log window</i> , page 260.
Interrupt Log Summary	Opens a window which displays a summary of the status of all defined interrupts, see <i>Interrupt Log Summary window</i> , page 264.
Memory Access Setup	Displays a dialog box to simulate memory access checking by specifying memory areas with different access types, see <i>Memory Access Setup dialog box</i> , page 157.
Trace	Opens a window which displays the collected trace data, see <i>Trace window</i> , page 184.
Function Trace	Opens a window which displays the trace data for function calls and function returns, see <i>Function Trace window</i> , page 189.
Function Profiler	Opens a window which shows timing information for the functions, see <i>Function Profiler window</i> , page 219.
Timeline	Opens a window which shows trace data for interrupt logs and for the call stack, see <i>Timeline window</i> , page 190.

Breakpoint Usage Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 121.

The C-SPY GDB Server driver

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

More specifically, this means:

- *GDB Server menu*, page 387.

GDB Server menu

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

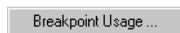


Figure 136: The GDB Server menu

This command is available on the menu:

Breakpoint Usage Opens a window which lists all active breakpoints; see *Breakpoint Usage window*, page 121.

The C-SPY J-Link/J-Trace driver

This section gives additional reference information for the C-SPY J-Link/J-Trace driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *J-Link menu*, page 388.

J-Link menu

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

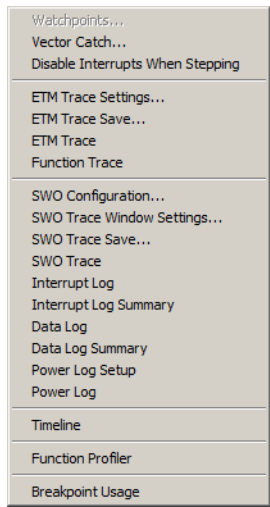


Figure 137: The J-Link menu

These commands are available on the menu:

Watchpoints	Displays a dialog box for setting watchpoints, see <i>Code breakpoints dialog box</i> , page 122.
Vector Catch	Displays a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see <i>Breakpoints on exception vectors</i> , page 112. Note that this command is not available for all ARM cores.
Disable Interrupts When Stepping	Ensures that only the stepped instructions will be executed. Interrupts will not be executed. This command can be used when not running at full speed and some interrupts interfere with the debugging process.
ETM Trace Settings¹	Displays a dialog box to configure ETM trace data generation and collection; see <i>ETM Trace Settings dialog box</i> , page 177.
ETM Trace Save¹	Displays a dialog box to save the collected trace data to a file; see <i>Trace Save dialog box</i> , page 188.

ETM Trace¹	Opens the ETM Trace window to display the collected trace data; see <i>Trace window</i> , page 184.
Function Trace¹	Opens a window which displays the trace data for function calls and function returns; see <i>Function Trace window</i> , page 189.
SWO Configuration²	Displays a dialog box; see <i>SWO Configuration dialog box</i> , page 181.
SWO Trace Window Settings²	Displays a dialog box; see <i>SWO Trace Window Settings dialog box</i> , page 179.
SWO Trace Save²	Displays a dialog box to save the collected trace data to a file; see <i>Trace Save dialog box</i> , page 188.
SWO Trace²	Opens the SWO Trace window to display the collected trace data; see <i>Trace window</i> , page 184.
Interrupt Log Log²	Opens a window; see <i>Interrupt Log window</i> , page 260.
Interrupt Log Summary²	Opens a window; see <i>Interrupt Log Summary window</i> , page 264.
Data Log²	Opens a window; see <i>Event Log window</i> , page 104.
Data Log Summary²	Opens a window; see <i>Data Log Summary window</i> , page 103.
Power Log Setup	Opens a window; see <i>Power Setup window</i> , page 234.
Power Log	Opens a window; see <i>Power Log window</i> , page 236.
Timeline³	Opens a window; see <i>Timeline window</i> , page 190.
Function Profiler	Opens a window which shows timing information for the functions; see <i>Function Profiler window</i> , page 219.
Breakpoint Usage	Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 121.

¹ Only available when using either ETM or J-Link with ETB.

² Only available when the SWD/SWO interface is used.

³ Available when using either ETM or SWD/SWO.

LIVE WATCH AND USE OF DCC

The following possibilities for using live watch apply:

For Cortex-M

Access to memory or setting breakpoints is always possible during execution. The DCC (Debug Communications Channel) unit is not available.

For ARMxxx-S devices

Setting hardware breakpoints is always possible during execution.

For ARM7/ARM9 devices, including ARMxxx-S

Memory accesses must be made by your application. By adding a small program—a *DCC handler*—that communicates with the debugger through the DCC unit to your application, memory can be read/written during execution. Software breakpoints can also be set by the DCC handler.

Just add the files `JLINKDCC_Process.c` and `JLINKDCC_HandleDataAbort.s` located in `arm\src\debugger\dcc` to your project and call the `JLINKDCC_Process` function regularly, for example every millisecond.

In your local copy of the `cstartup` file, modify the interrupt vector table so that data aborts will call the `JLINKDCC_HandleDataAbort` handler.

TERMINAL I/O AND USE OF DCC

The following possibilities for using Terminal I/O apply:

For Cortex-M

See *ITM Stimulus Ports*, page 183.

For ARM7/ARM9 devices, including ARMxxx-S

DCC can be used for Terminal I/O output by adding the file `arm\src\debugger\dcc\DCC_Write.c` to your project. `DCC_write.c` overrides the library function `write`. Functions such as `printf` can then be used to output text in real time to the C-SPY Terminal I/O window.

In this case, you can disable semihosting which means that the breakpoint it uses is freed for other purposes. To disable semihosting, choose **General Options>Library Configuration>Library low-level interface implementation>None**.

The C-SPY JTAGjet driver

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

More specifically, this means:

- *JTAGjet menu*, page 391.

JTAGjet menu

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

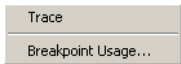


Figure 138: The JTAGjet menu

These commands are available on the menu:

Trace	Opens the JTAGjet Trace window; see <i>The JTAGjet Trace window</i> , page 162.
Breakpoint Usage	Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 121.

The C-SPY Macraigor driver

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

More specifically, this means:

- *Macraigor JTAG menu*, page 392.

Macraigor JTAG menu

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

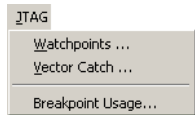


Figure 139: The Macraigor JTAG menu

These commands are available on the menu:

Watchpoints	Opens a dialog box for setting watchpoints, see <i>Code breakpoints dialog box</i> , page 122.
Vector Catch	Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see <i>Breakpoints on exception vectors</i> , page 112. Note that this command is not available for all ARM cores.
Breakpoint Usage	Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 121.

The C-SPY RDI driver

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

More specifically, this means:

- *RDI menu*, page 392.

RDI menu

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

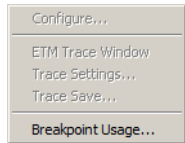


Figure 140: The RDI menu

These commands are available on the menu:

Configure	Opens a dialog box that originates from the RDI driver vendor. For information about details in this dialog box, refer to the driver documentation.
Trace Settings	Displays a dialog box to configure the ETM trace; see <i>ETM Trace Settings dialog box</i> , page 177.
Trace Save	Displays a dialog box to save the collected trace data to a file; see <i>Trace Save dialog box</i> , page 188.
Breakpoint Usage	Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 121.

Note: To get the default settings in the configuration dialog box, it is for some RDI drivers necessary to just open and close the dialog box even though you do not need any specific settings for your project.

The C-SPY ST-LINK driver

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

More specifically, this means:

- *ST-LINK menu*, page 393.

ST-LINK menu

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

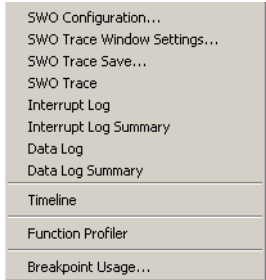


Figure 141: The ST-LINK menu

These commands are available on the menu:

SWO Configuration¹	Displays a dialog box; see <i>SWO Configuration dialog box</i> , page 181.
SWO Trace Window Settings¹	Displays a dialog box; see <i>SWO Trace Window Settings dialog box</i> , page 179.
SWO Trace Save¹	Displays a dialog box to save the collected trace data to a file; see <i>Trace Save dialog box</i> , page 188.
SWO Trace¹	Opens the SWO Trace window to display the collected trace data; see <i>Trace window</i> , page 184.
Interrupt Log¹	Opens a window; see <i>Interrupt Log window</i> , page 260.
Interrupt Log Summary¹	Opens a window; see <i>Interrupt Log Summary window</i> , page 264.
Data Log¹	Opens a window; see <i>Event Log window</i> , page 104.
Data Log Summary¹	Opens a window; see <i>Data Log Summary window</i> , page 103.
Timeline²	Opens a window; see <i>Timeline window</i> , page 190.
Function Profiler	Opens a window which shows timing information for the functions; see <i>Function Profiler window</i> , page 219.
Breakpoint Usage	Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 121.

¹ Only available when the SWD/SWO interface is used.

² Available when using either ETM or SWD/SWO.

The C-SPY TI Stellaris driver

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

More specifically, this means:

- *TI Stellaris menu*, page 395.

TI Stellaris menu

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



Figure 142: The TI Stellaris menu

This command is available on the menu:

Breakpoint Usage	Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 121.
-------------------------	---

The C-SPY TI XDS100 driver

This section gives additional reference information on the C-SPY TI XDS100 driver, reference information not provided elsewhere in this documentation.

More specifically, this means:

- *TI XDS100 menu*, page 395.

TI XDS100 menu

When you are using the C-SPY TI XDS100 driver, the **TI XDS100** menu is added to the menu bar.

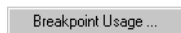


Figure 143: The TI XDS100 menu

This command is available on the menu:

Breakpoint Usage	Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 121.
-------------------------	---

Using flash loaders

This chapter describes the flash loader, what it is and how to use it. More specifically, this means:

- Introduction to the flash loader
- Reference information on the flash loader.

Introduction to the flash loader

This section introduces the flash loader.

This section covers these topics:

- Briefly about the flash loader
- Setting up the flash loader(s)
- The flash loading mechanism.

BRIEFLY ABOUT THE FLASH LOADER

A flash loader is an agent that is downloaded to the target. It fetches your application from the debugger and programs it into flash memory. The flash loader uses the file I/O mechanism to read the application program from the host. You can select one or several flash loaders, where each flash loader loads a selected part of your application. This means that you can use different flash loaders for loading different parts of your application.

A set of flash loaders for various microcontrollers is provided with IAR Embedded Workbench for ARM. In addition to these, more flash loaders are provided by chip manufacturers and third-party vendors. The flash loader API, documentation, and several implementation examples are available to make it possible for you to implement your own flash loader.

SETTING UP THE FLASH LOADER(S)

To use a flash loader for downloading your application:

- 1 Choose **Project>Options**.
- 2 Choose the **Debugger** category and click the **Download** tab.

- 3 Select the **Use Flash loader(s)** option. A default flash loader configured for the device you have specified will be used. The configuration is specified in a preconfigured board file.
- 4 To override the default flash loader or to modify the behavior of the default flash loader to suit your board, select the **Override default. board file** option, and **Edit** to open the **Flash Loader Configuration** dialog box. A copy of the *.board file will be created in your project directory and the path to the *.board file will be updated accordingly.
- 5 The **Flash Loader Overview** dialog box lists all currently configured flash loaders, see *Flash Loader Overview dialog box*, page 399. You can either select a flash loader or open the **Flash Loader Configuration** dialog box.

In the **Flash Loader Configuration** dialog box, you can configure the download. For more information about the various flash loader options, see *Flash Loader Configuration dialog box*, page 400.

THE FLASH LOADING MECHANISM

When the **Use flash loader(s)** option is selected and one or several flash loaders have been configured, the steps below are performed when the debug session starts:

Steps 1 to 4 are performed for each flash loader in the flash loader configuration.

- 1 C-SPY downloads the flash loader into target RAM.
- Steps 2 to 4 are performed one or more times depending on the size of the RAM and the size of the application image.
- 2 C-SPY writes code/data from the application image into target RAM (RAM buffer).
 - 3 C-SPY starts execution of the flash loader.
 - 4 The flash loader reads data from the RAM buffer and programs the flash memory.
 - 5 The application image now resides in flash memory and can be started. The flash loader and the RAM buffer are no longer needed, so RAM is fully available to the application in the flash memory.

Reference information on the flash loader

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

- *Flash Loader Overview dialog box*, page 399
- *Flash Loader Configuration dialog box*, page 400.

Flash Loader Overview dialog box

The **Flash Loader Overview** dialog box is available from the **Debugger>Download** page.

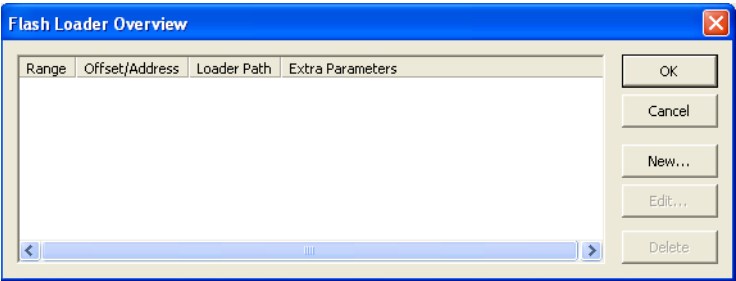


Figure 144: Flash Loader Overview dialog box

This dialog box lists all defined flash loaders. If you have selected a device on the **General Options>Target** page for which there is a flash loader, this flash loader is by default listed in the **Flash Loader Overview** dialog box.

The display area

Each row in the display area shows how you have set up one flash loader for flashing a specific part of memory:

Range	The part of your application to be programmed by the selected flash loader.
Offset/Address	The start of the memory where your application will be flashed. If the address is preceded with a, the address is absolute. Otherwise, it is a relative offset to the start of the memory.
Loader Path	The path to the flash loader *.flash file to be used (*.out for old-style flash loaders).
Extra Parameters	List of extra parameters that will be passed to the flash loader.

Click on the column headers to sort the list by range, offset/address, etc.

Function buttons

These function buttons are available:

OK	The selected flash loader(s) will be used for downloading your application to memory.
Cancel	Standard cancel.
New	Displays a dialog box where you can specify what flash loader to use, see <i>Flash Loader Configuration dialog box</i> , page 400.
Edit	Displays a dialog box where you can modify the settings for the selected flash loader, see <i>Flash Loader Configuration dialog box</i> , page 400.
Delete	Deletes the selected flash loader configuration.

Flash Loader Configuration dialog box

The **Flash Loader Configuration** dialog box is available from the **Flash Loader Overview** dialog box.

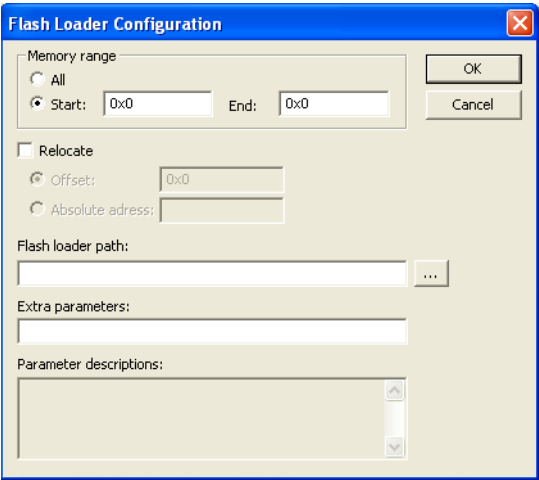


Figure 145: Flash Loader Configuration dialog box

Use the **Flash Loader Configuration** dialog box to configure the download to suit your board. A copy of the default `board` file will be created in your project directory.

Memory range

Specify the part of your application to be downloaded to flash memory. Choose between:

- All** The whole application is downloaded using this flash loader.
- Start/End** Specify the start and the end of the memory area for which part of the application will be downloaded.

Relocate

Overrides the default flash base address, that is relocate the location of the application in memory. This means that you can flash your application to a different location from where it was linked. Choose between:

- Offset** A numeric value for a relative offset. This offset will be added to the addresses in the application file.
- Absolute address** A numeric value for an absolute base address where the application will be flashed. The lowest address in the application will be placed on this address. Note that you can only use one flash loader for your application when you specify an absolute address.

You can use these numeric formats:

- 123456 Decimal numbers.
- 0x123456 Hexadecimal numbers
- 0123456 Octal numbers

The default base address used for writing the first byte—the lowest address—to flash is specified in the linker configuration file used for your application. However, it can sometimes be necessary to override the flash base address and start at a different location in the address space. This can, for example, be necessary for devices that remap the location of the flash memory.

Flash loader path

Use the text box to specify the path to the flash loader file (`*.flash`) to be used by your board configuration.

Extra parameters

Some flash loaders define their own set of specific options. Use this text box to specify options to control the flash loader. For information about available flash loader options, see the **Parameter descriptions** field.

Parameter descriptions

The **Parameter descriptions** field displays a description of the extra parameters specified in the **Extra parameters** text box.

A

absolute location, specifying for a breakpoint. 136

Access Type (Data breakpoints option). 125, 129

Access Type (Data Log breakpoints option). 131

Access type (Edit Memory Access option). 159

Access Type (Immediate breakpoints option). 134

Access type (Trace Filter option). 206

Access type (Trace Start option). 201

Access type (Trace Stop option). 204

Action (Code breakpoints option). 123

Action (Immediate breakpoints option). 134

Actual (SWO clock setting). 183

Adaptive (JTAG/SWD speed setting). 368

Add to Watch Window (Symbolic Memory window context menu). 152

Add (Watch window context menu). 94

Address Bus Pattern (Address setting). 125

Address Range (Find in Trace option). 210

Address (JTAG Watchpoints option). 125

Allow hardware reset (JTAGjet option). 371

Allow hardware reset (RDI option). 377

Ambiguous symbol (Resolve Symbol Ambiguity option). 100

Angel (debugger option). 355

Any Size (Data setting). 125

Any (Access Type setting). 125

Any (Extern setting). 126

Any (Mode setting). 126

Append to file (Trace Save option). 189

application, built outside the IDE 48

assembler labels, viewing 88

assembler source code, fine-tuning 213

assembler symbols, using in C-SPY expressions 85

assembler variables, viewing. 88

assumptions, programming experience. 25

Attach to program (debugger option). 357

Auto Scroll (Timeline window context menu). 195

Auto window 92

Auto (Default breakpoint type setting). 133

Auto (JTAG/SWD speed setting). 368

Auto (Size setting). 131

Auto (SWO setting). 369, 380

Autodetect (SWO clock setting). 183

Autostep settings dialog box. 81

Autostep (Debug menu). 59

B

--backend (C-SPY command line option). 327

backtrace information

- generated by compiler 69
- viewing in Call Stack window 75

batch mode, using C-SPY in. 321

Baud rate (Macraigor option). 376

Baud rate (Serial port settings option). 362, 364

--BE32 (C-SPY command line option). 323

--BE8 (C-SPY command line option). 323

Big Endian (Memory window context menu). 145

blocks, in C-SPY macros 278

bold style, in this guide. 29

Break At (Code breakpoints option). 122

Break At (Data breakpoints option). 129

Break At (Immediate breakpoints option). 134

Break At (Log breakpoints option). 127

Break Condition (JTAG Watchpoints option). 126

Break on Throw (Debug menu). 59

Break on Uncaught Exception (Debug menu). 59

Break (Debug menu). 58

breakpoint condition, example 117–118

Breakpoint type (Code breakpoints option). 122

Breakpoint Usage window 121

Breakpoint Usage (JTAGjet menu). 391

Breakpoint Usage (J-Link menu). 389

Breakpoint Usage (Macraigor JTAG menu). 392

Breakpoint Usage (RDI menu). 393

Breakpoint Usage (Simulator menu). 387

Breakpoint Usage (ST-LINK menu). 394

Breakpoint Usage (TI Stellaris menu). 395

Breakpoint Usage (TI XDS100 menu)	395
breakpoints	
code, example	307
connecting a C-SPY macro	274
consumers of	111
data	128
example	308
data log	130
description of	108
disabling used by Stack window	112
icons for in the IDE	110
in Memory window	115
listing all	121
log, example	310
profiling source	214, 220
reasons for using	107
setting	
in memory window	115
using system macros	116
using the dialog box	114
single-stepping if not available	47
toggling	113
trace start, example	313
trace stop, example	314
types of	108
useful tips	117
Breakpoints dialog box	
Code	122
Data	128
Data Log	130
Immediate	134
Log	127
Trace Filter (J-Link)	205
Trace Start	198, 200
Trace Stop	199
Breakpoints options (C-SPY options)	132
Breakpoints window	119
Broadcast all branches (ETM Trace Settings option)	178
Browse (Trace toolbar)	185

byte order, setting in Memory window	145
Byte (Data setting)	125

C

C function information, in C-SPY	69
C symbols, using in C-SPY expressions	85
C variables, using in C-SPY expressions	85
call chain, displaying in C-SPY	69
Call stack information	69
Call Stack window	75
for backtrace information	70
Call Stack (Timeline window context menu)	195
__cancelAllInterrupts (C-SPY system macro)	284
__cancelInterrupt (C-SPY system macro)	284
Catch exceptions (Breakpoints option)	133
Catch exceptions (JTAGjet option)	372
Catch exceptions (RDI option)	378
Chain (Break Condition setting)	127
Clear All (Debug Log window context menu)	79
Clear trace data (Trace toolbar)	185
Clear (Interrupt Log window context menu)	263
Clear (Power Log window context menu)	238
__clearBreak (C-SPY system macro)	285
Clock setup (J-Link/J-Trace option)	369
Clock setup (ST-LINK option)	380
__closeFile (C-SPY system macro)	285
code breakpoints, overview	108
Code Coverage window	242
Code Coverage (Disassembly window context menu)	73
--code_coverage_file (C-SPY command line option)	328
code, covering execution of	242
command line options	327
typographic convention	29
command prompt icon, in this guide	29
communication problem, J-Link	368
Communication (Angel option)	362
Communication (J-Link/J-Trace option)	370
computer style, typographic convention	29

- conditional statements, in C-SPY macros 277
- Conditions (Code breakpoints option) 123
- Conditions (Log breakpoints option) 128
- Configure (RDI menu) 393
- Connect during reset (Reset setting) 365, 379
- context menu, in windows. 88
- conventions, used in this guide 28
- Copy Window Contents (Disassembly window context menu) 74
- Copy (Debug Log window context menu) 79
- copyright notice 2
- Core and peripherals (Reset setting) 365
- Core (Reset setting) 365
- CPI (Generate setting) 180
- cpu (C-SPY command line option). 324
- CPU clock (SWO Configuration option). 182
- CPU clock (SWO setting). 369, 380
- cspybat 321
- current position, in C-SPY Disassembly window 72
- cursor, in C-SPY Disassembly window. 72
- Cycle accurate tracing (ETM Trace Settings option). 178
- cycles (C-SPY command line option) 328
- C-SPY
 - batch mode, using in 321
 - debugger systems, overview of 35
 - differences between drivers 38
 - environment overview 31
 - plugin modules, loading. 48
 - setting up 46–47
 - starting the debugger 48
- C-SPY drivers
 - overview 37
 - specifying 355
- C-SPY expressions 84
 - evaluating. 98
 - in C-SPY macros 277
 - Tooltip watch, using. 83
 - Watch window, using 83
- C-SPY hardware drivers, hardware installation. 43
- C-SPY macro "__message"
 - style (Log breakpoints option) 128
- C-SPY macros
 - blocks. 278
 - conditional statements 277
 - C-SPY expressions 277
 - dialog box, using 271
 - examples 269
 - checking status of register. 273
 - checking the status of WDT 273
 - creating a log macro 274
 - execUserPreload, using. 52
 - remapping memory before download 52
 - executing 270
 - connecting to a breakpoint 274
 - using Quick Watch 273
 - using setup macro and setup file 272
 - functions 86, 276
 - loop statements 278
 - macro statements 277
 - setup macro file 268
 - executing. 272
 - setup macro functions 268
 - summary 280
 - system macros, summary of. 281
 - using 267
 - variables. 86, 276
- C-SPY options 353
 - Extra Options. 358
 - Images 359
 - Plugins. 360
 - Setup 355
- C-SPYLink 37
- C++ exceptions
 - debugging 59
 - single stepping. 66
- C++ terminology. 28

D

data breakpoints, overview	109
Data Bus Pattern (Data setting)	126
Data Coverage (Memory window context menu)	146
data coverage, in Memory window	144
data log breakpoints, overview	109
Data Log Events (SWO Configuration option)	182
Data Log Summary window	103
Data Log Summary (J-Link menu)	389
Data Log Summary (ST-LINK menu)	394
Data Log window	101
Data Log (J-Link menu)	389
Data Log (ST-LINK menu)	394
Data Log (Timeline window context menu)	195
Data value + exact addr (Data Log Events setting)	182
Data (JTAG Watchpoints option)	125
DCC (Debug Communications Channel)	390
ddf (filename extension), selecting a file	47
Debug handler address (Macraigor option)	377
Debug Log window	79
Debug Log window context menu	79
Debug menu (C-SPY main window)	58
debugger concepts, definitions of	34
debugger system overview	35
debugging projects	
externally built applications	48
loading multiple images	50
debugging, RTOS awareness	33
Default breakpoint type (Breakpoints option)	133
__delay (C-SPY system macro)	285
Delay (Autostep Settings option)	81
Delete (Breakpoints window context menu)	120
Description (Edit Interrupt option)	256
description (interrupt property)	256
--device (C-SPY command line option)	328
Device description file (debugger option)	356
device description files	47
definition of	51

modifying	51
specifying interrupts	301
Disable All (Breakpoints window context menu)	120
Disable Interrupts When Stepping (J-Link menu)	388
Disable (Breakpoints window context menu)	120
__disableInterrupts (C-SPY system macro)	286
--disable_interrupts (C-SPY command line option)	329
Disassemble in ARM mode (Disassembly menu)	60
Disassemble in Auto mode (Disassembly menu)	60
Disassemble in Current processor mode (Disassembly menu)	60
Disassemble in Thumb mode (Disassembly menu)	60
Disassembly menu (C-SPY main window)	60
Disassembly window	71
context menu	73
disclaimer	2
DLIB, documentation	27
do (macro statement)	278
document conventions	28
documentation	
overview of guides	27
overview of this guide	26
this guide	25
--download_only (C-SPY command line option)	329
Driver (debugger option)	355
__driverType (C-SPY system macro)	286
--drv_attach_to_program (C-SPY command line option)	324
--drv_catch_exceptions (C-SPY command line option)	330
--drv_communication (C-SPY command line option)	331
--drv_communication_log (C-SPY command line option)	333
--drv_default_breakpoint (C-SPY command line option)	334
--drv_reset_to_cpu_start (C-SPY command line option)	334
--drv_restore_breakpoints (C-SPY command line option)	335
--drv_suppress_download (C-SPY command line option)	324
--drv_swo_clock_setup (C-SPY command line option)	335
--drv_vector_table_base (C-SPY command line option)	336
--drv_verify_download (C-SPY command line option)	324

E

Edit Expressions (Trace toolbar)	186
Edit Interrupt dialog box	256
Edit Memory Access dialog box	159
Edit Settings (Trace toolbar)	186
Edit (Breakpoints window context menu)	120
edition, of this guide	2
Embedded C++ Technical Committee	28
EmbeddedICE macrocell	109
__emulatorSpeed (C-SPY system macro)	287
__emulatorStatusCheckOnRead (C-SPY system macro)	287
Enable All (Breakpoints window context menu)	120
Enable interrupt simulation (Interrupt Setup option)	254
Enable Log File (Log File option)	80
Enable (Breakpoints window context menu)	120
Enable (Interrupt Log window context menu)	263
Enable (Power Log window context menu)	237
Enable (Timeline window context menu)	195
Enabled ports (ITM Stimulus Ports setting)	183
__enableInterrupts (C-SPY system macro)	288
Enable/Disable Breakpoint (Call Stack window context menu)	76
Enable/Disable Breakpoint (Disassembly window context menu)	74
Enable/Disable (Trace toolbar)	185
End address (Edit Memory Access option)	159
--endian (C-SPY command line option)	324
endianness. <i>See</i> byte order	
Enter Location dialog box	135
ETM Configuration dialog box	168
ETM Control dialog box	167
ETM Trace Save (J-Link menu)	388
ETM Trace Settings (J-Link menu)	388
ETM Trace (J-Link menu)	389
ETM/ETB (J-Link/J-Trace option)	369
__evaluate (C-SPY system macro)	288
Event Log Summary window	105
Event Log window	104

examples

C-SPY macros	269
interrupts,timer.	251
macros	
checking status of register	273
checking status of WDT	273
creating a log macro	274
using Quick Watch	273
performing tasks and continue execution	118
tracing incorrect function arguments	117
EXC (Generate setting)	180
execUserExit (C-SPY setup macro)	281
execUserFlashExit (C-SPY setup macro)	281
execUserFlashInit (C-SPY setup macro)	280
execUserFlashReset (C-SPY setup macro)	280
execUserPreload (C-SPY setup macro)	280
execUserPreReset (C-SPY setup macro)	280
execUserReset (C-SPY setup macro)	281
execUserSetup (C-SPY setup macro)	280
executed code, covering	242
execution history, tracing	176
Expression (Access Type setting)	131
expressions. <i>See</i> C-SPY expressions	
Extend to cover requested range (Trigger range setting)	132
Extend to cover requested range (Trigger range setting)	129, 201, 204, 206
Extern (JTAG Watchpoints option)	126
Extra Options, for C-SPY	358

F

File format (Memory Save option)	147
file types	
device description, specifying in IDE	47
macro	47, 356
File (Trace Save option)	189
filename extensions	
ddf, selecting device description file	47
mac, using macro file	47

Filename (Memory Restore option)	148
Filename (Memory Save option).	147
Fill dialog box.	148
Find in Trace dialog box	209
Find in Trace window	210
Find (Memory window context menu)	146
Find (Trace toolbar)	185
first activation time (interrupt property) definition of	247
First activation (Edit Interrupt option).	257
Fixed (JTAG/SWD speed setting).	368
flash loader parameters to control	402
specifying the path to.	401
using	397
Flash Loader Overview dialog box	399
flash memory, load library module to	298
--flash_loader (C-SPY command line option).	337
FOLD (Generate setting)	180
for (macro statement)	278
Force (SWO Trace Window Settings option)	179
Forced Interrupt window.	257
Forced Interrupts (Simulator menu)	386
--fpu (C-SPY command line option).	325
Frequency in Hz (Driver1 option).	362, 364–366, 368
Function Profiler window	219
Function Profiler (J-Link menu)	389
Function Profiler (Simulator menu)	386
Function Profiler (ST-LINK menu).	394
Function Trace window.	189
Function Trace (GDB Server menu)	387
Function Trace (J-Link menu).	389
Function Trace (Simulator menu)	386
functions, C-SPY running to when starting.	46, 356

G

GDB Server (C-SPY driver), menu.	387
GDB Server (debugger option)	355

__gdbserver_exec_command (C-SPY system macro).	289
--gdbserv_exec_command (C-SPY command line option)	337
Generate (SWO Trace Window Settings option).	180
Go to Source (Breakpoints window context menu).	120
Go to Source (Call Stack window context menu)	76
Go To Source (Timeline window context menu).	196
Go (Debug menu)	58, 68

H

Halfword (Data setting)	125
Halt after bootloader (Reset setting)	366
Halt before bootloader (Reset setting).	366
Hardware reset (Macraigor option).	376
hardware setup, power consumption because of	231
Hardware (Default breakpoint type setting)	133
Hardware, Atmel AT91SAM7 (Reset setting).	367
Hardware, halt after delay (ms) (Reset setting).	366
Hardware, halt at 0 (Reset setting)	367
Hardware, halt using Breakpoint (Reset setting).	367
Hardware, halt using DBGRQ (Reset setting).	367
Hardware, NXP LPC (Reset setting).	367
highlighting, in C-SPY	69
Hold time (Edit Interrupt option)	257
hold time (interrupt property), definition of	247
__hwReset (C-SPY system macro).	289
__hwResetRunToBp (C-SPY system macro)	290
__hwResetWithStrategy (C-SPY system macro)	291

I

IAR debugger driver plugin (debugger option).	382
IAR ROM-monitor (debugger option)	355
icons, in this guide	29
if else (macro statement).	278
if (macro statement)	277
illegal memory accesses, checking for	142
Images window.	62
Images, loading multiple.	359

immediate breakpoints, overview	109
In use by (Data Log Events setting)	182
In use by (PC Sampling setting)	181
Include (Log File option)	80
Index Range (Trace Save option)	188
Input Mode dialog box	78
input, special characters in Terminal I/O window	77
installation directory	28
Instruction Profiling (Disassembly window context menu)	74
Intel-extended, C-SPY output format	36
Interface (J-Link/J-Trace option)	370
Interface (Macraigor option)	375
Interface (ST-LINK option)	379
Interface (TI Stellaris option)	381
interference, power consumption because of	231
interrupt handling, power consumption during	230
Interrupt Log Summary window	264
Interrupt Log Summary (J-Link menu)	389
Interrupt Log Summary (Simulator menu)	386
Interrupt Log Summary (ST-LINK menu)	394
Interrupt Log window	260
Interrupt Log (J-Link menu)	389
Interrupt Log (Simulator menu)	386
Interrupt Log (ST-LINK menu)	394
Interrupt Log (SWO Configuration option)	182
Interrupt Logs (Force setting)	179
Interrupt Setup dialog box	254
Interrupt Setup (Simulator menu)	386
Interrupt Status window	258
interrupt system, using device description file	249
Interrupt (Edit Interrupt option)	256
Interrupt (Timeline window context menu)	195
interrupts	
adapting C-SPY system for target hardware	249
simulated, introduction to	245
timer, example	251
using system macros	249
__isBatchMode (C-SPY system macro)	292
italic style, in this guide	29

ITM Stimulus Ports (SWO Configuration option)	183
---	-----

J

__jlinkExecCommand (C-SPY system macro)	292
--jlink_device_select (C-SPY command line option)	337
--jlink_exec_command (C-SPY command line option)	338
--jlink_initial_speed (C-SPY command line option)	338
--jlink_interface (C-SPY command line option)	339
--jlink_ir_length (C-SPY command line option)	339
--jlink_reset_strategy (C-SPY command line option)	340
--jlink_script_file (C-SPY command line option)	340
--jlink_speed (C-SPY command line option)	341
--jlink_trace_source (C-SPY command line option)	341
JTAG interfaces, J-Link	365, 369
JTAG scan chain with multiple targets (Macraigor option)	376
JTAG scan chain with multiple targets	
(JTAG scan chain setting)	371
JTAG scan chain (J-Link/J-Trace option)	371
JTAG speed (Macraigor option)	375
JTAG Watchpoints dialog box	124
JTAG watchpoints, overview	109
JTAG (Interface setting)	370, 375, 379, 381
__jtagCommand (C-SPY system macro)	292
__jtagCP15IsPresent (C-SPY system macro)	293
__jtagCP15ReadReg (C-SPY system macro)	293
__jtagCP15WriteReg (C-SPY system macro)	294
__jtagData (C-SPY system macro)	294
JTAGjet Trace window	162
JTAGjet (C-SPY driver)	
menu	391
JTAGjet (debugger option)	355
__jtagRawRead (C-SPY system macro)	295
__jtagRawSync (C-SPY system macro)	295
__jtagRawWrite (C-SPY system macro)	296
__jtagResetTRST (C-SPY system macro)	297
JTAG/SWD speed (J-Link/J-Trace option)	368
JTAG/SWD speed (TI Stellaris option)	381
J-Link communication problem	368

J-Link JTAG interface	365, 369
J-Link (C-SPY driver), menu	388
J-Link/J-Trace (debugger option)	355

L

labels (assembler), viewing	88
Length (Fill option).	149
lightbulb icon, in this guide.	29
Link condition (Trace Filter option)	207
Link condition (Trace Start option).	202
Link condition (Trace Stop option).	205
Little Endian (Memory window context menu)	145
Live Watch window	94
--lmiftdi_speed (C-SPY command line option).	342
__loadImage (C-SPY system macro)	297
loading multiple debug files, list currently loaded.	62
loading multiple images	50
Locals window	92
log breakpoints, overview	108
Log communication (Angel option)	362–363
Log communication (debugger option).	383
Log communication (IAR ROM-monitor option).	364
Log communication (JTAGjet option).	372
Log communication (J-Link/J-Trace option).	371
Log communication (Macraigor option).	377
Log communication (TI Stellaris option)	381
Log File dialog box.	80
Log RDI communication (RDI option).	378
Logging>Set Log file (Debug menu)	60
Logging>Set Terminal I/O Log file (Debug menu).	60
loop statements, in C-SPY macros	278
low-power mode, power consumption during	228
LSU (Generate setting)	180

M

mac (filename extension), using a macro file	47
Macraigor (C-SPY driver), menu	392

Macraigor (debugger option)	355
--macro (C-SPY command line option)	342
Macro Configuration dialog box	271
macro files, specifying	47, 356
macro statements	277
macros	270
executing	270
using	267
Macros (Debug menu)	60
--mac_handler_address (C-SPY command line option)	342
--mac_interface (C-SPY command line option)	343
--mac_jtag_device (C-SPY command line option)	343
--mac_multiple_targets (C-SPY command line option)	344
--mac_reset_pulls_reset (C-SPY command line option)	344
--mac_set_temp_reg_buffer (C-SPY command line option)	344
--mac_speed (C-SPY command line option).	345
--mac_xscale_ir7 (C-SPY command line option)	345
main function, C-SPY running to when starting	46, 356
Manual (Size setting)	131
Manufacturer RDI driver (RDI option)	377
--mapu (C-SPY command line option)	346
Mask (Address setting)	125
Mask (Data setting).	126
Mask (Match data setting)	130, 202, 204, 207
Match data (Data breakpoints option).	130
Match data (Trace Filter option)	207
Match data (Trace Start option)	202
Match data (Trace Stop option).	204
memory access checking.	142
Memory access checking (Memory Access Setup option)	158
Memory Access Setup dialog box.	157
Memory Access Setup (Simulator menu)	386
memory accesses, illegal.	142
Memory Fill (Memory window context menu).	146
memory map.	157
Memory range (Edit Memory Access option).	159
Memory Restore dialog box	148
Memory Restore (Memory window context menu).	146
Memory Save dialog box	147

Memory Save (Memory window context menu) 146
 Memory window 143
 memory zones 141
 __memoryRestore (C-SPY system macro) 298
 __memorySave (C-SPY system macro) 299
 Memory>Restore (Debug menu) 59
 Memory>Save (Debug menu) 59
 menu bar, C-SPY-specific 57
 Message (Log breakpoints option) 128
 migration, from earlier IAR compilers 27
 MISRA C, documentation 27
 Mixed Mode (Disassembly window context menu) 75
 Mode (JTAG Watchpoints option) 126
 Motorola, C-SPY output format 36
 Move to PC (Disassembly window context menu) 73
 Multi-ICE interface. *See* RealView Multi-ICE interface

N

naming conventions 29
 Navigate (Timeline window context menu) 194
 New Breakpoint (Breakpoints window context menu) . . . 121
 Next Statement (Debug menu) 59
 Next Symbol (Symbolic Memory window context menu) 151
 Non User (Mode setting) 126
 Normal (Break Condition setting) 126
 Normal (Reset setting) 365, 379
 Normal, disable watchdog (Reset setting) 366

O

OCD interface device (Macraigor option) 375
 OP Fetch (Access Type setting) 125
 __openFile (C-SPY system macro) 300
 Operation (Fill option) 149
 operators, sizeof in C-SPY 86
 optimizations, effects on variables 87
 options
 in the IDE 353

 on the command line 327, 358
 Options (Stack window context menu) 154
 __orderInterrupt (C-SPY system macro) 301
 Originator (debugger option) 360
 Override default .board file (debugger option) 358
 Override project default (SWO Configuration option) . . 182

P

-p (C-SPY command line option) 346
 parameters
 list of passed to the flash loader 399
 tracing incorrect values of 69
 typographic convention 29
 part number, of this guide 2
 PC only (Data Log Events setting) 182
 PC samples (Force setting) 179
 PC Sampling (SWO Configuration option) 181
 PC + data value + base addr (Data Log Events setting) . . 182
 peripheral units
 detecting unattended 229
 device-specific 51
 peripheral units, in Register window 140
 Please select one symbol
 (Resolve Symbol Ambiguity option) 100
 --plugin (C-SPY command line option) 346
 plugin modules (C-SPY) 36
 loading 48
 Plugins (C-SPY options) 360
 __popSimulatorInterruptExecutingStack (C-SPY
 system macro) 302
 pop-up menu. *See* context menu
 Port (Macraigor option) 376
 Port (Serial port settings option) 362, 364
 power consumption, measuring 214, 225
 Power Log Setup (J-Link menu) 389
 Power Log window 236
 Power Log (J-Link menu) 389
 Power Log (Timeline window context menu) 195

power sampling.	214
Power Setup window	234
Preceding bits (JTAG scan chain setting)	371
prerequisites, programming experience.	25
Previous Symbol (Symbolic Memory window context menu)	151
probability (interrupt property)	257
definition of.	247
Probability % (Edit Interrupt option)	257
Probe configuration (JTAGjet option)	372
--proc_stack_xxx (C-SPY command line option)	347
Profile Selection (Timeline window context menu)	196
profiling	
on function level	216
on instruction level.	216
profiling information, on functions and instructions	214
profiling sources	
breakpoints	214, 220
sampling	214, 221
trace (calls)	214, 220
trace (flat)	214, 221
program execution, in C-SPY	65
programming experience.	25
program, <i>see also</i> application	
projects, for debugging externally built applications.	48
publication date, of this guide.	2

Q

Quick Watch window	98
executing C-SPY macros	273

R

Range for (Viewing Range option)	197
Range (Break Condition setting).	127
Rate (PC Sampling setting).	182
RDI Configuration dialog box for JTAGjet.	373
RDI (C-SPY driver), menu	392

RDI (debugger option)	355
--rdi_allow_hardware_reset (C-SPY command line option)	347
--rdi_driver_dll (C-SPY command line option).	348
--rdi_heartbeat (C-SPY command line option)	325
--rdi_step_max_one (C-SPY command line option)	348
Read (Access Type setting).	125, 131
__readFile (C-SPY system macro)	302
__readFileByte (C-SPY system macro)	303
reading guidelines.	25
__readMemoryByte (C-SPY system macro).	303
__readMemory8 (C-SPY system macro)	303
__readMemory16 (C-SPY system macro)	304
__readMemory32 (C-SPY system macro)	304
RealView Multi-ICE interface	377
reference information, typographic convention.	29
Refresh (Debug menu)	59
register groups	140
predefined, enabling.	155
Register window	155
registered trademarks	2
__registerMacroFile (C-SPY system macro).	304
registers, displayed in Register window	155
Remove (Watch window context menu)	94
Repeat interval (Edit Interrupt option)	257
repeat interval (interrupt property), definition of.	247
Replace (Memory window context menu)	146
Reset Pin (Reset setting).	365, 379
Reset (Debug menu)	58
Reset (J-Link/J-Trace option)	365, 379
__resetFile (C-SPY system macro).	305
Resolve Source Ambiguity dialog box	137
Restore software breakpoints at (Breakpoints option).	133
Restore (Memory Restore option).	148
__restoreSoftwareBreakpoints (C-SPY system macro).	305
return (macro statement).	278
ROM-monitor, definition of	36
RTOS awareness debugging	33
RTOS awareness (C-SPY plugin module).	34
Run to Cursor (Call Stack window context menu)	76

Run to Cursor (Debug menu)	59
Run to Cursor (Disassembly window context menu)	73
Run to Cursor, command for executing.	69
Run to (C-SPY option)	46
Run to (debugger option)	356
R/W (Access Type setting)	125

S

sampling, profiling source.	214, 221
Save to log file (Interrupt Log window context menu)	263
Save to log file (Power Log window context menu)	238
Save (Memory Save option)	147
Save (Trace toolbar)	185
Scale (Viewing Range option).	198
Scan chain contains non-ARM devices (JTAG scan chain setting)	371
Select All (Debug Log window context menu)	79
Select Graphs (Timeline window context menu).	196
Select plugins to load (debugger option).	360
Semihosted, SWI (option), using	77
--semihosting (C-SPY command line option)	349
Send heartbeat (Angel option)	362
Serial port settings (Angel option)	362
Serial port settings (IAR ROM-monitor option)	364
Set Data Breakpoint (Memory window context menu)	146
Set Next Statement (Debug menu)	59
Set Next Statement (Disassembly window context menu)	74
__setCodeBreak (C-SPY system macro).	306
__setDataBreak (C-SPY system macro)	307
__setLogBreak (C-SPY system macro)	309
__setSimBreak (C-SPY system macro)	311
__setTraceStartBreak (C-SPY system macro).	312
__setTraceStopBreak (C-SPY system macro).	314
setup macro functions	268
reserved names.	280
Setup macros (debugger option)	356
Setup (C-SPY options)	355
SFR	
in Register window	156
using as assembler symbols	85
shortcut menu. <i>See</i> context menu	
Show all images (Images window context menu)	63
Show Arguments (Call Stack window context menu)	76
Show As (Watch window context menu)	94
Show Cycles (Interrupt Log window context menu)	263
Show Cycles (Power Log window context menu).	238
Show Numerical Value (Timeline window context menu)	196
Show offsets (Stack window context menu)	154
Show only (Image window context menu)	63
Show Time (Interrupt Log window context menu)	263
Show Time (Power Log window context menu)	238
Show timestamp (ETM Trace Settings option)	178
Show variables (Stack window context menu)	154
--silent (C-SPY command line option)	349
simulating interrupts, enabling/disabling	254
simulator driver, selecting.	39
Simulator menu.	386
Simulator (debugger option)	355
simulator, introduction	39
Size (Code breakpoints option).	123
Size (Data Log breakpoints option)	131
Size (Timeline window context menu)	196
Size (Trace Filter option)	206
Size (Trace Start option)	201
Size (Trace Stop option)	203
sizeof	86
SLEEP (Generate setting)	180
software delay, power consumption during	227
Software (Default breakpoint type setting)	133
Software (Reset setting)	367
Software, Analog devices (Reset setting)	367
Solid Graph (Timeline window context menu)	196
__sourcePosition (C-SPY system macro)	315
special function registers (SFR)	
in Register window	156
using as assembler symbols	85

stack usage, computing	142
Stack window	152
stack.mac	268
Stall processor on FIFO full (ETM Trace Settings option)	178
standard C, sizeof operator in C-SPY	86
Start address (Edit Memory Access option)	159
Start address (Fill option)	149
Start address (Memory Save option)	147
Statics window	95
stdin and stdout, redirecting to C-SPY window	77
Step Into (Debug menu)	58
Step Into, description	67
Step Out (Debug menu)	59
Step Out, description	68
Step Over (Debug menu)	58
Step Over, description	67
step points, definition of	66
--stlink_interface (C-SPY command line option)	349
--stlink_reset_strategy (C-SPY command line option)	350
Stop address (Memory Save option)	147
Stop Debugging (Debug menu)	58
__strFind (C-SPY system macro)	315
ST-LINK (C-SPY driver), menu	393
ST-LINK (debugger option)	355
__subString (C-SPY system macro)	316
Suppress download (debugger option)	357, 382
SWD interface, information in Trace window	172
SWD (Interface setting)	370, 375, 379, 381
SWO clock (SWO Configuration option)	183
SWO communication channel	
enabling	370, 375, 381
for timestamps in trace	179
SWO Configuration dialog box	181
SWO Configuration (J-Link menu)	389
SWO Configuration (ST-LINK menu)	394
SWO Trace Save (J-Link menu)	389
SWO Trace Save (ST-LINK menu)	394
SWO Trace Settings dialog box	179
SWO Trace Window Settings (J-Link menu)	389

SWO Trace Window Settings (ST-LINK menu)	394
SWO Trace (J-Link menu)	389
SWO Trace (ST-LINK menu)	394
Symbolic Memory window	150
Symbols window	99
symbols, using in C-SPY expressions	84

T

TAP number (JTAG scan chain setting)	371
target system, definition of	35
__targetDebuggerVersion (C-SPY system macro)	316
TCP/IP address or hostname (GDB Server option)	363
TCP/IP Macraigor option)	376
TCP/IP (Angel option)	362
TCP/IP (Communication setting)	370
Terminal IO Log Files (Terminal IO Log Files option)	78
Terminal I/O Log Files dialog box	78
Terminal I/O window	70, 77
terminology	28
Text search (Find in Trace option)	209
Third-Party Driver (debugger options)	382
TI Stellaris (C-SPY driver), menu	395
TI Stellaris (debugger option)	355
TI XDS100 (C-SPY driver), menu	395
Time Axis Unit (Timeline window context menu)	196
time interval, in Timeline window	217
Time Stamps (Force setting)	179
Timeline window	190
Timeline (J-Link menu)	389
Timeline (Simulator menu)	386
Timeline (ST-LINK menu)	394
--timeout (C-SPY command line option)	350
timer interrupt, example	251
timestamps in SWO trace	179
Timestamps (SWO Configuration option)	183
To Log File (ITM Stimulus Ports setting)	183
To Terminal I/O window (ITM Stimulus Ports setting)	183

Toggle Breakpoint (Code) (Call Stack window context menu) 76
 Toggle Breakpoint (Code) (Disassembly window context menu) 74
 Toggle Breakpoint (Log) (Call Stack window context menu) 76
 Toggle Breakpoint (Log) (Disassembly window context menu) 74
 Toggle Breakpoint (Trace Start) (Disassembly window context menu) 74
 Toggle Breakpoint (Trace Stop) (Disassembly window context menu) 74
 Toggle source (Trace toolbar) 185
 __toLower (C-SPY system macro) 316
 tools icon, in this guide 29
 __toString (C-SPY system macro) 317
 __toUpper (C-SPY system macro) 317
 Trace buffer size (Trace Settings option) 178
 Trace Expressions window 208
 Trace Filter breakpoints dialog box (J-Link) 205
 Trace port mode (Trace Settings option) 177
 Trace port width (Trace Settings option) 177
 Trace Save dialog box 188
 Trace Save (RDI menu) 393
 Trace search query dialog box 165
 Trace Settings dialog box 177
 Trace Settings (RDI menu) 393
 trace start and stop breakpoints, overview 108
 Trace Start breakpoints dialog box 198, 200
 Trace Stop breakpoints dialog box 199
 Trace view field configuration dialog box 164
 Trace window 184
 trace (calls), profiling source 214, 220
 trace (flat), profiling source 214, 221
 Trace (JTAGjet menu) 391
 Trace (Simulator menu) 386
 trace, in Timeline window 190
 trademarks 2
 Trigger at (Data Log breakpoints option) 131
 Trigger at (Trace Filter option) 206

Trigger at (Trace Start option) 200
 Trigger at (Trace Stop option) 203
 Trigger range (Data breakpoints option) 129
 Trigger range (Data Log breakpoints option) 132
 Trigger range (Trace Filter option) 206
 Trigger range (Trace Start option) 201
 Trigger range (Trace Stop option) 204
 Trigger (Forced Interrupt window context menu) 258
 typographic conventions 29

U

Unavailable, C-SPY message 87
 __unloadImage(C-SPY system macro) 318
 USB (Communication setting) 370
 Use command line options (debugger option) 358
 Use Extra Images (debugger option) 359
 Use flash loader (debugger option) 358
 Use manual ranges (Memory Access Setup option) 158
 Use ranges based on (Memory Access Setup option) 157
 Use tab-separated format (Trace Save option) 189
 user application, definition of 35
 User (Mode setting) 126

V

Value (Address setting) 125
 Value (Data setting) 126
 Value (Fill option) 149
 Value (Match data setting) 130, 202, 204, 207
 variables
 effects of optimizations 87
 information, limitation on 86
 using in C-SPY expressions 85
 variance (interrupt property), definition of 247
 Variance % (Edit Interrupt option) 257
 Vector Catch dialog box 135
 Vector Catch (Macraigor JTAG menu) 392
 Verify all (debugger option) 382

Verify download (debugger option)	357
version number, of this guide	2
Viewing Range dialog box	197
Viewing Range (Timeline window context menu)	196
visualSTATE, C-SPY plugin module for	37

W

waiting for device, power consumption during	227
Wanted (SWO clock setting)	183
warnings icon, in this guide	29
Watch window	93
using	83
Watchpoints (J-Link menu)	388
Watchpoints (Macraigor JTAG menu)	392
web sites, recommended	28
while (macro statement)	278
windows, specific to C-SPY	61
Word (Data setting)	125
Write (Access Type setting)	125, 131
__writeFile (C-SPY system macro)	318
__writeFileByte (C-SPY system macro)	319
__writeMemoryByte (C-SPY system macro)	319
__writeMemory8 (C-SPY system macro)	319
__writeMemory16 (C-SPY system macro)	320
__writeMemory32 (C-SPY system macro)	320

X

--xds_rootdir (C-SPY command line option)	351
---	-----

Z

Zone (Edit Memory Access option)	159
Zone (Fill option)	149
Zone (Memory Restore option)	148
Zone (Memory Save option)	147
Zone (Memory window context menu)	145
zone, in C-SPY	141

Zoom (Timeline window context menu)	195
---	-----

Symbols

__cancelAllInterrupts (C-SPY system macro)	284
__cancelInterrupt (C-SPY system macro)	284
__clearBreak (C-SPY system macro)	285
__closeFile (C-SPY system macro)	285
__delay (C-SPY system macro)	285
__disableInterrupts (C-SPY system macro)	286
__driverType (C-SPY system macro)	286
__emulatorSpeed (C-SPY system macro)	287
__emulatorStatusCheckOnRead (C-SPY system macro)	287
__enableInterrupts (C-SPY system macro)	288
__evaluate (C-SPY system macro)	288
__fmessage (C-SPY macro statement)	278
__gdbserver_exec_command (C-SPY system macro)	289
__hwReset (C-SPY system macro)	289
__hwResetRunToBp (C-SPY system macro)	290
__hwResetWithStrategy (C-SPY system macro)	291
__isBatchMode (C-SPY system macro)	292
__jlinkExecCommand (C-SPY system macro)	292
__jtagCommand (C-SPY system macro)	292
__jtagCP15IsPresent (C-SPY system macro)	293
__jtagCP15ReadReg (C-SPY system macro)	293
__jtagCP15WriteReg (C-SPY system macro)	294
__jtagData (C-SPY system macro)	294
__jtagRawRead (C-SPY system macro)	295
__jtagRawSync (C-SPY system macro)	295
__jtagRawWrite (C-SPY system macro)	296
__jtagResetTRST (C-SPY system macro)	297
__loadImage (C-SPY system macro)	297
__memoryRestore (C-SPY system macro)	298
__memorySave (C-SPY system macro)	299
__message (C-SPY macro statement)	278
__openFile (C-SPY system macro)	300
__orderInterrupt (C-SPY system macro)	301
__popSimulatorInterruptExecutingStack (C-SPY system macro)	302

__readFile (C-SPY system macro)	302	--download_only (C-SPY command line option)	329
__readFileByte (C-SPY system macro)	303	--drv_attach_to_program (C-SPY command line option)	324
__readMemoryByte (C-SPY system macro)	303	--drv_catch_exceptions (C-SPY command line option) . .	330
__readMemory8 (C-SPY system macro)	303	--drv_communication (C-SPY command line option) . . .	331
__readMemory16 (C-SPY system macro)	304	--drv_communication_log (C-SPY command line option)	333
__readMemory32 (C-SPY system macro)	304	--drv_default_breakpoint (C-SPY command line option) .	334
__registerMacroFile (C-SPY system macro)	304	--drv_reset_to_cpu_start (C-SPY command line option) .	334
__resetFile (C-SPY system macro)	305	--drv_restore_breakpoints (C-SPY command line option)	335
__restoreSoftwareBreakpoints (C-SPY system macro) . .	305	--drv_suppress_download (C-SPY command line option)	324
__setCodeBreak (C-SPY system macro)	306	--drv_swo_clock_setup (C-SPY command line option)	335
__setDataBreak (C-SPY system macro)	307	--drv_vector_table_base (C-SPY command line option) . .	336
__setLogBreak (C-SPY system macro)	309	--drv_verify_download (C-SPY command line option) . .	324
__setSimBreak (C-SPY system macro)	311	--endian (C-SPY command line option)	324
__setTraceStartBreak (C-SPY system macro)	312	--flash_loader (C-SPY command line option)	337
__setTraceStopBreak (C-SPY system macro)	314	--fpu (C-SPY command line option)	325
__smessage (C-SPY macro statement)	278	--gdbserve_exec_command (C-SPY command line option)	337
__sourcePosition (C-SPY system macro)	315	--jlink_device_select (C-SPY command line option) . . .	337
__strFind (C-SPY system macro)	315	--jlink_exec_command (C-SPY command line option) .	338
__subString (C-SPY system macro)	316	--jlink_initial_speed (C-SPY command line option) . . .	338
__targetDebuggerVersion (C-SPY system macro)	316	--jlink_interface (C-SPY command line option)	339
__toLower (C-SPY system macro)	316	--jlink_ir_length (C-SPY command line option)	339
__toString (C-SPY system macro)	317	--jlink_reset_strategy (C-SPY command line option) . . .	340
__toUpper (C-SPY system macro)	317	--jlink_script_file (C-SPY command line option)	340
__unloadImage (C-SPY system macro)	318	--jlink_speed (C-SPY command line option)	341
__writeFile (C-SPY system macro)	318	--jlink_trace_source (C-SPY command line option)	341
__writeFileByte (C-SPY system macro)	319	--lmi_ftdi_speed (C-SPY command line option)	342
__writeMemoryByte (C-SPY system macro)	319	--macro (C-SPY command line option)	342
__writeMemory8 (C-SPY system macro)	319	--mac_handler_address (C-SPY command line option) . .	342
__writeMemory16 (C-SPY system macro)	320	--mac_interface (C-SPY command line option)	343
__writeMemory32 (C-SPY system macro)	320	--mac_jtag_device (C-SPY command line option)	343
-p (C-SPY command line option)	346	--mac_multiple_targets (C-SPY command line option) . .	344
--backend (C-SPY command line option)	327	--mac_reset_pulls_reset (C-SPY command line option) . .	344
--BE32 (C-SPY command line option)	323	--mac_set_temp_reg_buffer (C-SPY command line option)	344
--BE8 (C-SPY command line option)	323		
--code_coverage_file (C-SPY command line option) . . .	328		
--cpu (C-SPY command line option)	324		
--cycles (C-SPY command line option)	328		
--device (C-SPY command line option)	328		
--disable_interrupts (C-SPY command line option)	329		

--mac_speed (C-SPY command line option)	345
--mac_xscale_ir7 (C-SPY command line option)	345
--mapu (C-SPY command line option)	346
--plugin (C-SPY command line option)	346
--proc_stack_xxx (C-SPY command line option)	347
--rdi_allow_hardware_reset (C-SPY command line option)	347
--rdi_driver_dll (C-SPY command line option).	348
--rdi_heartbeat (C-SPY command line option)	325
--rdi_step_max_one (C-SPY command line option)	348
--semihosting (C-SPY command line option)	349
--silent (C-SPY command line option)	349
--stlink_interface (C-SPY command line option)	349
--stlink_reset_strategy (C-SPY command line option) . .	350
--timeout (C-SPY command line option)	350
--xds_rootdir (C-SPY command line option)	351

Numerics

1x Units (Memory window context menu)	145
1x Units (Stack window context menu)	154
1x Units (Symbolic Memory window context menu)	151
2x Units (Memory window context menu)	145
2x Units (Stack window context menu)	154
2x Units (Symbolic Memory window context menu)	151
4x Units (Memory window context menu)	145
4x Units (Stack window context menu)	154
4x Units (Symbolic Memory window context menu)	152
8x Units (Memory window context menu)	145