

**IAR Embedded
Workbench**

C-SPY® Debugging Guide

for
RISC-V

UCSRISCV-12



COPYRIGHT NOTICE

© 2019–2023 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, Embedded Trust, C-Trust, IAR Connect, C-SPY, C-RUN, C-STAT, IAR Visual State, IAR KickStart Kit, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR Systems are trademarks or registered trademarks owned by IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

RISC-V is a registered trademark of RISC-V International.

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

Twelfth edition: October 2023

Part number: UCSRISCV-12

This guide applies to version 3.30.x of IAR Embedded Workbench® for RISC-V.

Internal reference: BB15, FF9.2.5, tut2009.1, ISHP

Brief contents

Tables	19
Preface	21
Part 1. Basic debugging	27
The IAR C-SPY Debugger	29
Getting started using C-SPY	41
Executing your application	57
Variables and expressions	77
Breakpoints	105
Memory and registers	129
Part 2. Analyzing your application	173
Trace	175
The application timeline	197
Profiling	215
Code coverage	225
Part 3. Advanced debugging	231
Multicore debugging	233
Interrupts	243
C-SPY macros	267
The C-SPY command line utility— <code>cspybat</code>	333
Flash loaders	365

Part 4. Additional reference information	371
Debugger options	373
Additional information on C-SPY drivers	391
Index	401

Contents

Tables	19
Preface	21
Who should read this guide	21
Required knowledge	21
How to use this guide	21
What this guide contains	22
Part 1. Basic debugging	22
Part 2. Analyzing your application	22
Part 3. Advanced debugging	23
Part 4. Additional reference information	23
Other documentation	23
User and reference guides	23
The online help system	24
Web sites	24
Document conventions	24
Typographic conventions	25
Naming conventions	26
Part I. Basic debugging	27
The IAR C-SPY Debugger	29
Introduction to C-SPY	29
An integrated environment	29
General C-SPY debugger features	30
RTOS awareness	31
Debugger concepts	31
C-SPY and target systems	32
The debugger	33
The target system	33
The application	33
C-SPY debugger systems	33

Third-party debuggers	34
C-SPY plugin modules	34
C-SPY drivers overview	35
Differences between the C-SPY drivers	35
The IAR C-SPY Simulator	36
Supported features	36
The C-SPY hardware debugger drivers	36
Communication overview	37
Hardware installation	38
USB driver installation	38
Getting started using C-SPY	41
Setting up C-SPY	41
Setting up for debugging	41
Executing from reset	42
Using a setup macro file	42
Selecting a device description file	43
Loading plugin modules	43
Starting C-SPY	43
Starting a debug session	44
Loading executable files built outside of the IDE	44
Starting a debug session with source files missing	44
Loading multiple debug images	45
Editing in C-SPY windows	46
Adapting for target hardware	47
Modifying a device description file	47
Initializing target hardware before C-SPY starts	48
Reference information on starting C-SPY	48
C-SPY Debugger main window	49
Images window	53
Get Alternative File dialog box	54
Executing your application	57
Introduction to application execution	57
Briefly about application execution	57

Source and disassembly mode debugging	57
Single stepping	58
Troubleshooting slow stepping speed	60
Running the application	61
Highlighting	62
Viewing the call stack	63
Terminal input and output	63
Debug logging	64
Reference information on application execution	64
Disassembly window	65
Call Stack window	70
Terminal I/O window	72
Terminal I/O Log File dialog box	73
Debug Log window	73
Report Assert dialog box	75
Autostep settings dialog box	75
Variables and expressions	77
Introduction to working with variables and expressions	77
Briefly about working with variables and expressions	77
C-SPY expressions	78
Limitations on variable information	80
Working with variables and expressions	81
Using the windows related to variables and expressions	81
Viewing assembler variables	82
Reference information on working with variables and expressions	83
Auto window	83
Locals window	86
Watch window	88
Live Watch window	91
Statics window	94
Quick Watch window	97
Symbols window	100

Resolve Symbol Ambiguity dialog box	103
Breakpoints	105
Introduction to setting and using breakpoints	105
Reasons for using breakpoints	105
Briefly about setting breakpoints	105
Breakpoint types	106
Breakpoint icons	108
Breakpoints in the C-SPY simulator	108
Breakpoints in the C-SPY hardware debugger drivers	108
Breakpoint consumers	109
Setting breakpoints	110
Various ways to set a breakpoint	110
Toggling a simple code breakpoint	110
Setting breakpoints using the dialog box	110
Setting a data breakpoint in the Memory window	112
Setting breakpoints using system macros	112
Useful breakpoint hints	113
Reference information on breakpoints	115
Breakpoints window	115
Breakpoint Usage window	117
Code breakpoints dialog box	118
Log breakpoints dialog box	120
Data breakpoints dialog box (Simulator)	121
Data breakpoints dialog box (for C-SPY hardware debugger drivers)	123
Data Log breakpoints dialog box	124
Immediate breakpoints dialog box	125
Enter Location dialog box	126
Resolve Source Ambiguity dialog box	128
Memory and registers	129
Introduction to monitoring memory and registers	129
Briefly about monitoring memory and registers	129
C-SPY memory zones	130

Memory configuration for the C-SPY simulator	131
Memory configuration for C-SPY hardware debugger drivers	132
Monitoring memory and registers	133
Defining application-specific register groups	133
Monitoring stack usage	134
Reference information on memory and registers	137
Memory window	138
Memory Save dialog box	142
Memory Restore dialog box	143
Fill dialog box	144
Symbolic Memory window	145
Stack window	148
Registers window	152
Register User Groups Setup window	155
SFR Setup window	157
Edit SFR dialog box	160
Memory Configuration dialog box for the C-SPY simulator	162
Edit Memory Range dialog box for the C-SPY simulator	164
Memory Configuration dialog box for C-SPY hardware debugger drivers	166
Edit Memory Range dialog box for C-SPY hardware debugger drivers	169
 Part 2. Analyzing your application	 173
Trace	175
Introduction to using trace	175
Reasons for using trace	175
Briefly about trace	175
Requirements for using trace	177
Collecting and using trace data	177
Getting started with trace	177
Trace data collection using breakpoints	178

Searching in trace data	178
Browsing through trace data	179
Reference information on trace	179
Trace Settings button in the IDE toolbar	179
Trace Settings dialog box	180
Trace window	183
Function Trace window	190
Trace Start Trigger breakpoint dialog box	191
Trace Stop Trigger breakpoint dialog box	193
Find in Trace dialog box	194
Find in Trace window	195
The application timeline	197
Introduction to analyzing your application's timeline	197
Briefly about analyzing the timeline	197
Requirements for timeline support	198
Analyzing your application's timeline	198
Displaying a graph in the Timeline window	199
Navigating in the graphs	199
Analyzing performance using the graph data	199
Getting started using data logging	201
Reference information on application timeline	201
Data Log window	202
Data Log Summary window	205
Timeline window—Call Stack graph	207
Viewing Range dialog box	212
Profiling	215
Introduction to the profiler	215
Reasons for using the profiler	215
Briefly about the profiler	215
Requirements for using the profiler	216
Using the profiler	216
Getting started using the profiler on function level	217
Analyzing the profiling data	217

Getting started using the profiler on instruction level	219
Reference information on the profiler	220
Function Profiler window	220
Code coverage	225
Introduction to code coverage	225
Reasons for using code coverage	225
Briefly about code coverage	225
Requirements and restrictions for using code coverage	225
Using code coverage	226
Getting started using code coverage	226
Reference information on code coverage	226
Code Coverage window	227
Part 3. Advanced debugging	231
Multicore debugging	233
Introduction to multicore debugging	233
Briefly about multicore debugging	233
Symmetric multicore debugging	233
Asymmetric multicore debugging	234
Requirements and restrictions for multicore debugging	235
Debugging multiple cores	235
Setting up for symmetric multicore debugging	235
Setting up for asymmetric multicore debugging	235
Starting and stopping a multicore debug session	238
Reference information on multicore debugging	238
Cores window	239
Multicore toolbar	241
The multicore session file	241
Interrupts	243
Introduction to interrupts	243
Briefly about the interrupt simulation system	243
Interrupt characteristics	244

Interrupt simulation states	245
C-SPY system macros for interrupt simulation	246
Target-adapting the interrupt simulation system	247
Briefly about interrupt logging	247
Using the interrupt system	247
Simulating a simple interrupt	248
Simulating an interrupt in a multi-task system	249
Getting started using interrupt logging	250
Reference information on interrupts	250
Interrupt Configuration window	251
Available Interrupts window	254
Interrupt Status window	255
Interrupt Log window	257
Interrupt Log Summary window	260
Timeline window—Interrupt Log graph	263
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	268
Using C-SPY macros	269
Registering C-SPY macros—an overview	270
Executing C-SPY macros—an overview	270
Registering and executing using setup macros and setup files	271
Executing macros using Quick Watch	271
Executing a macro by connecting it to a breakpoint	272
Aborting a C-SPY macro	273
Reference information on the macro language	274
Macro functions	274
Macro variables	274
Macro parameters	275
Macro strings	275

Macro statements	276
Formatted output	277
Reference information on reserved setup macro function names	279
execUserAttach	280
execUserPreload	280
execUserExecutionStarted	280
execUserExecutionStopped	281
execUserFlashInit	281
execUserSetup	281
execUserFlashReset	282
execUserPreReset	282
execUserReset	282
execUserExit	282
execUserFlashExit	283
execUserCoreConnect	283
Reference information on C-SPY system macros	283
__abortLaunch	286
__cancelAllInterrupts	287
__cancelInterrupt	287
__clearBreak	287
__closeFile	288
__delay	288
__disableInterrupts	289
__driverType	289
__enableInterrupts	290
__evaluate	290
__fillMemory8	291
__fillMemory16	291
__fillMemory32	292
__fillMemory64	293
__gdbserver_exec_command	294
__getNumberOfCores	295
__getSelectedCore	295

__isBatchMode	296
__isMacroSymbolDefined	296
__loadImage	297
__memoryRestore	298
__memorySave	299
__messageBoxYesCancel	300
__messageBoxYesNo	301
__openFile	301
__orderInterrupt	302
__popSimulatorInterruptExecutingStack	303
__probeType	304
__readFile	304
__readFileByte	305
__readMemory8, __readMemoryByte	306
__readMemory16	306
__readMemory32	307
__readMemory64	307
__registerMacroFile	308
__resetFile	308
__selectCore	308
__setCodeBreak	309
__setDataBreak	310
__setDataLogBreak	312
__setLogBreak	313
__setSimBreak	314
__setTraceStartBreak	315
__setTraceStopBreak	316
__sourcePosition	317
__strFind	317
__subString	318
__system1	319
__system2	319
__system3	320
__targetDebuggerVersion	321

__toLower	321
__toString	322
__toUpper	322
__unloadImage	323
__wallTime_ms	323
__writeFile	324
__writeFileByte	324
__writeMemory8, __writeMemoryByte	325
__writeMemory16	325
__writeMemory32	326
__writeMemory64	326
Graphical environment for macros	327
Macro Registration window	327
Debugger Macros window	329
Macro Quicklaunch window	331
The C-SPY command line utility— <code>cspybat</code>	333
Using C-SPY in batch mode	333
Starting <code>cspybat</code>	333
Output	334
Invocation syntax	334
Summary of C-SPY command line options	335
General <code>cspybat</code> options	335
Options available for all C-SPY drivers	336
Options available for the simulator driver	336
Options available for all hardware debugger drivers	337
Options available for the C-SPY I-jet driver	337
Options available for the C-SPY GDB Server driver	338
Options available for the C-SPY third-party drivers	338
Reference information on C-SPY command line options ..	338
--application_args	338
--attach_to_running_target	339
--backend	340
--code_coverage_file	340

--core	341
--cycles	341
--debug_file	342
--device_macro	342
--disable_interrupts	342
--disable_misalignment_exception	343
--download_only	343
--drv_catch_exceptions	344
--drv_communication	345
--drv_communication_log	346
--drv_default_breakpoint	346
--drv_exclude_from_verify	347
--drv_interface	347
--drv_interface_speed	348
--drv_reset_to_cpu_start	348
--drv_restore_breakpoints	348
--drv_suppress_download	349
--drv_system_bus_access	349
--drv_vector_table_base	350
--drv_verify_download	350
-f	350
--flash_loader	351
--function_profiling	351
--gdbserv_exec_command	352
--jet_board_cfg	352
--jet_board_did	353
--jet_ir_length	353
--jet_itc_output	354
--jet_power_from_probe	354
--jet_script_file	355
--jet_sigprobe_opt	355
--jet_standard_reset	357
--jet_startup_connection_timeout	358
--jet_tap_position	358

--leave_target_running	359
--macro	359
--macro_param	360
--mapu	360
--multicore_nr_of_cores	361
-p	361
--plugin	361
--reset_style	362
--silent	363
--suppress_entrypoint_warning	363
--timeout	364
Flash loaders	365
Introduction to the flash loader	365
Using flash loaders	365
Setting up the flash loader(s)	365
Aborting a flash loader	366
Reference information on the flash loader	366
Flash Loader Overview dialog box	366
Flash Loader Configuration dialog box	368
Part 4. Additional reference information	371
Debugger options	373
Setting debugger options	373
Reference information on general debugger options	374
Setup	374
Download	375
Images	376
Multicore	377
Extra Options	379
Plugins	380

Reference information on C-SPY hardware debugger driver options	381
GDB Server: Setup	381
GDB Server: Breakpoints	382
I-jet : Setup	383
I-jet : Interface	386
I-jet : Breakpoints	388
Third-Party Driver options	389
Additional information on C-SPY drivers	391
Reference information on C-SPY driver menus	391
<i>C-SPY driver</i>	391
Simulator menu	392
Reference information on the C-SPY simulator	394
Data Alignment Setup dialog box	394
Simulated Frequency dialog box	395
Reference information on the C-SPY hardware debugger drivers	395
GDB Server menu	396
I-jet menu	396
Resolving problems	398
Write failure during load	398
No contact with the target hardware	399
Index	401

Tables

1: Typographic conventions used in this guide	25
2: Naming conventions used in this guide	26
3: Driver differences	35
4: C-SPY assembler symbols expressions	79
5: Handling name conflicts between hardware registers and assembler labels	79
6: C-SPY macros for breakpoints	113
7: Support for timeline information	198
8: C-SPY driver profiling support	216
9: Project options for enabling the profiler	217
10: Project options for enabling code coverage	226
11: Timer interrupt settings	249
12: Examples of C-SPY macro variables	275
13: Summary of system macros	283
14: __cancelInterrupt return values	287
15: __disableInterrupts return values	289
16: __driverType return values	289
17: __enableInterrupts return values	290
18: __evaluate return values	290
19: __isBatchMode return values	296
20: __loadImage return values	297
21: __messageBoxYesCancel return values	300
22: __messageBoxYesNo return values	301
23: __openFile return values	302
24: __probeType return values	304
25: __readFile return values	305
26: __setCodeBreak return values	310
27: __setDataBreak return values	311
28: __setDataLogBreak return values	312
29: __setLogBreak return values	313
30: __setSimBreak return values	314
31: __setTraceStartBreak return values	315

32: __setTraceStopBreak return values	316
33: __sourcePosition return values	317
34: __unloadImage return values	323
35: cspybat parameters	334
36: Options specific to the C-SPY drivers you are using	373

Preface

Welcome to the *C-SPY® Debugging Guide for RISC-V*. 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 RISC-V.

Who should read this guide

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

REQUIRED KNOWLEDGE

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

- The architecture and instruction set of the RISC-V core you are using (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation*, page 23.

How to use this guide

Each chapter in this guide covers a specific *topic area*. In many chapters, information is typically divided into different sections based on *information types*:

- *Concepts*, which describes the topic and gives overviews of features related to the topic area. Any requirements or restrictions are also listed. Read this section to learn about the topic area.
- *Tasks*, which lists useful tasks related to the topic area. For many of the tasks, you can also find step-by-step descriptions. Read this section for information about required tasks as well as for information about how to perform certain tasks.
- *Reference information*, which gives reference information related to the topic area. Read this section for information about certain features or GUI components. You can easily access this type of information for a GUI component in the IDE by pressing F1.

If you are new to using IAR Embedded Workbench, we suggest that you first go through the tutorials, which you can find in IAR Information Center in the product, under **Product Explorer**. They will help you get started.

Finally, we recommend the *Glossary* if you should encounter any unfamiliar terms in the IAR user documentation.

What this guide contains

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

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

PART I. BASIC DEBUGGING

- *The IAR C-SPY Debugger* introduces you to the C-SPY debugger and to the concepts that are related to debugging in general and to C-SPY in particular. The chapter also introduces the various C-SPY drivers. The chapter briefly shows the difference in functionality that the various C-SPY drivers provide.
- *Getting started using C-SPY* helps you get started using C-SPY, which includes setting up, starting, and adapting C-SPY for target hardware.
- *Executing your application* describes the conceptual differences between source and disassembly mode debugging, the facilities for executing your application, and finally, how you can handle terminal input and output.
- *Variables and expressions* describes the syntax of the expressions and variables used in C-SPY, as well as the limitations on variable information. The chapter also demonstrates the various methods for monitoring variables and expressions.
- *Breakpoints* describes the breakpoint system and the various ways to set breakpoints.
- *Memory and registers* shows how you can examine memory and registers.

PART 2. ANALYZING YOUR APPLICATION

- *Trace* describes how you can inspect the program flow up to a specific state using trace data.
- *The application timeline* describes the **Timeline** window, and how to use the information in it to analyze your application's behavior.
- *Profiling* describes how the profiler can help you find the functions in your application source code where the most time is spent during execution.

- *Code coverage* describes how the code coverage functionality can help you verify whether all parts of your code have been executed, thus identifying parts which have not been executed.

PART 3. ADVANCED DEBUGGING

- *Multicore debugging* describes how to debug a target with multiple cores.
- *Interrupts* contains detailed information about the C-SPY interrupt simulation system and how to configure the simulated interrupts to make them reflect the interrupts of your target hardware.
- *C-SPY macros* describes the C-SPY macro system, its features, the purposes of these features, and how to use them.
- *The C-SPY command line utility—cspybat* describes how to use C-SPY in batch mode.
- *Flash loaders* describes the flash loader, what it is and how to use it.

PART 4. ADDITIONAL REFERENCE INFORMATION

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

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the IAR 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 development tools is described in a series of guides. Information about:

- System requirements and information about how to install and register the IAR products are available in the *Installation and Licensing Quick Reference Guide* and the *Licensing Guide*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for RISC-V*.
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for RISC-V*.

- Programming for the IAR C/C++ Compiler for RISC-V and linking, is available in the *IAR C/C++ Development Guide for RISC-V*.
- Programming for the IAR Assembler for RISC-V, is available in the *IAR Assembler User Guide for RISC-V*.
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Using I-jet, refer to the *IAR Debug probes User Guide for I-jet®*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains information about:

- IDE project management and building
- Debugging using the IAR C-SPY® Debugger
- The IAR C/C++ Compiler and Linker
- The IAR Assembler
- C-STAT

WEB SITES

Recommended web sites:

- The chip manufacturer's web site.
- The RISC-V International web site, www.riscv.org, that contains information and news about the RISC-V ISA. This includes the most recent specifications.
- The IAR 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.
- The C++ programming language web site, isocpp.org. This web site also has a list of recommended books about C++ programming.
- The C and C++ reference web site, en.cppreference.com.

Document conventions

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

When referring to a directory in your product installation, for example `riscv\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\riscv\doc`, where the initial digit of the version number reflects the initial digit of the version number of the IAR Embedded Workbench shared components.

TYPOGRAPHIC CONVENTIONS

The IAR documentation set 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 <code>filename.h</code> where <i>filename</i> represents the name of the file.
[option]	An optional part of a linker or stack usage control directive, where [and] are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
{option}	A mandatory part of a linker or stack usage control directive, where { and } are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
[option]	An optional part of a command line option, pragma directive, or library filename.
[a b c]	An optional part of a command line option, pragma directive, or library filename with alternatives.
{a b c}	A mandatory part of a command line option, pragma directive, or library filename 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.

Table 1: Typographic conventions used in this guide

Style	Used for
	Identifies warnings.

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

NAMING CONVENTIONS

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

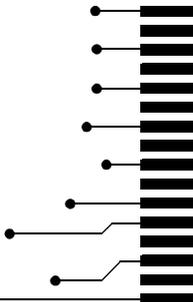
Brand name	Generic term
IAR Embedded Workbench® for RISC-V	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for RISC-V	the IDE
IAR C-SPY® Debugger for RISC-V	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for RISC-V	the compiler
IAR Assembler™ for RISC-V	the assembler
IAR ILINK Linker™	ILINK, the linker
IAR DLIB Runtime Environment™	the DLIB runtime environment

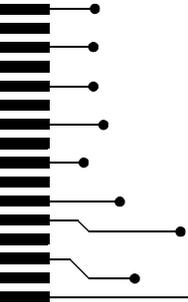
Table 2: *Naming conventions used in this guide*

Part I. Basic debugging

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

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





The IAR C-SPY Debugger

- Introduction to C-SPY
- Debugger concepts
- C-SPY drivers overview
- The IAR C-SPY Simulator
- The C-SPY hardware debugger drivers

Introduction to C-SPY

These topics are covered:

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

AN INTEGRATED ENVIRONMENT

C-SPY is a high-level-language debugger for embedded applications. It is designed for use with the IAR compilers and assemblers, and is completely integrated in the IDE, providing development and debugging within the same application. This gives 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 IAR 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 provides an entire toolchain, the output from the compiler and linker can include extensive debug information for the debugger, resulting in good debugging possibilities for you.

C-SPY offers these general features:

- *Source and disassembly level debugging*
C-SPY allows you to switch between source and disassembly debugging as required, for both C or C++ and assembler source code.
- *Single-stepping on a function call level*
Compared to traditional debuggers, where the finest granularity for source level stepping is line by line, C-SPY provides a finer level of control by identifying every statement and function call as a step point. This means that each function call—inside expressions, and function calls that are part of parameter lists to other functions—can be single-stepped. The latter is especially useful when debugging C++ code, where numerous extra function calls are made, for example to object constructors.
- *Code and data breakpoints*
The C-SPY breakpoint system lets you set breakpoints of various kinds in the application being debugged, allowing you to stop at locations of particular interest. For example, you set breakpoints to investigate whether your program logic is correct or to investigate how and when the data changes.
- *Monitoring variables and expressions*
For variables and expressions there is a wide choice of facilities. You can easily monitor values of a specified set of variables and expressions, continuously or on demand. You can also choose to monitor only local variables, static variables, etc.
- *Container awareness*
When you run your application in C-SPY, you can view the elements of library data types such as STL lists and vectors. This gives you a very good overview and debugging opportunities when you work with C++ STL containers.
- *Call stack information*
The compiler generates extensive call stack information. This allows the debugger to show, without any runtime penalty, the complete stack of function calls wherever the program counter is. You can select any function in the call stack, and for each function you get valid information for local variables and available registers.
- *Powerful macro system*
C-SPY includes a powerful internal macro system, to allow you to define complex sets of actions to be performed. C-SPY macros can be used on their own or in

conjunction with complex breakpoints and—for some cores or devices—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 RTOS-aware debugging. For information about which operating systems that are currently supported, see the Information Center, available from the **Help** menu.

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

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

A loaded plugin will add its own menu, set of windows, and buttons when a debug session is started (provided that the RTOS is linked with the application). For information about other RTOS awareness plugin modules, refer to the manufacturer of the plugin module.

Debugger concepts

This section introduces some of the concepts and terms that are related to debugging in general and to C-SPY in particular. This section does not contain specific information related to C-SPY features. Instead, you will find such information in the other chapters

of this documentation. The IAR user documentation uses the terms described in this section when referring to these concepts.

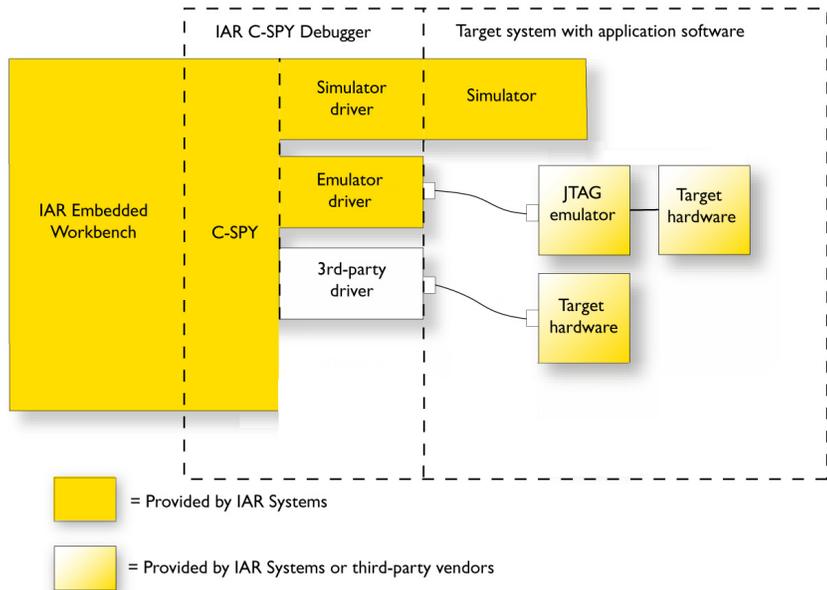
These topics are covered:

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

C-SPY AND TARGET SYSTEMS

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

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



THE DEBUGGER

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

THE TARGET SYSTEM

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

THE APPLICATION

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

C-SPY DEBUGGER SYSTEMS

C-SPY consists of both a general part which provides a basic set of debugger features, and a target-specific back end. The back end consists of two components: a processor

module—one for every microcontroller, which defines the properties of the microcontroller, and a *C-SPY driver*. The C-SPY driver is the part that provides communication with and control of the target system. The driver also provides the user interface—menus, windows, and dialog boxes—to the functions provided by the target system, for instance, special breakpoints.

Typically, there are three main types of C-SPY drivers:

- Simulator driver
- ROM-monitor driver
- Emulator driver

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

THIRD-PARTY DEBUGGERS

You can use a third-party debugger together with the IAR 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, or can be supplied by third-party vendors. Examples of such modules are:

- The various C-SPY drivers for debugging using certain debug systems.
- RTOS plugin modules for support for real-time OS aware debugging.
- C-SPYLink that bridges IAR Visual State and IAR Embedded Workbench to make true high-level state machine debugging possible directly in C-SPY, in addition to the normal C level symbolic debugging. For more information, see the documentation provided with IAR Visual State.

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

C-SPY drivers overview

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

- Simulator
- I-jet and I-jet Trace debug probes
- GDB Server

Note: In addition to the drivers supplied with IAR Embedded Workbench, you can also load debugger drivers supplied by a third-party vendor; see *Third-Party Driver options*, page 389.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

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

Feature	Simulator	I-jet	GDB Server
Code breakpoints ¹	Unlimited	Yes	Yes
Data breakpoints ¹	Yes	Yes	Yes
Execution in real time	—	Yes	Yes ²
Zero memory footprint	Yes	Yes	Yes ²
Simulated interrupts	Yes	—	—
Real interrupts	—	Yes	—
Interrupt logging	Yes	—	—
Data logging	Yes	—	—
Live watch	Yes	Yes	—
Cycle counter	Yes	—	—
Code coverage ¹	Yes	Yes	—
Data coverage	Yes	—	—
Function/instruction profiling ¹	Yes	Yes	—
Trace	Yes	Yes	—
Multicore debugging	Yes	Yes	—

Table 3: Driver differences

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

2. If connected to hardware.

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.

SUPPORTED FEATURES

The C-SPY simulator supports:

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

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

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

The C-SPY hardware debugger drivers

C-SPY can connect to a hardware debugger using a C-SPY hardware debugger driver as an interface. Any C-SPY hardware debugger drivers are automatically installed during the installation of IAR Embedded Workbench.

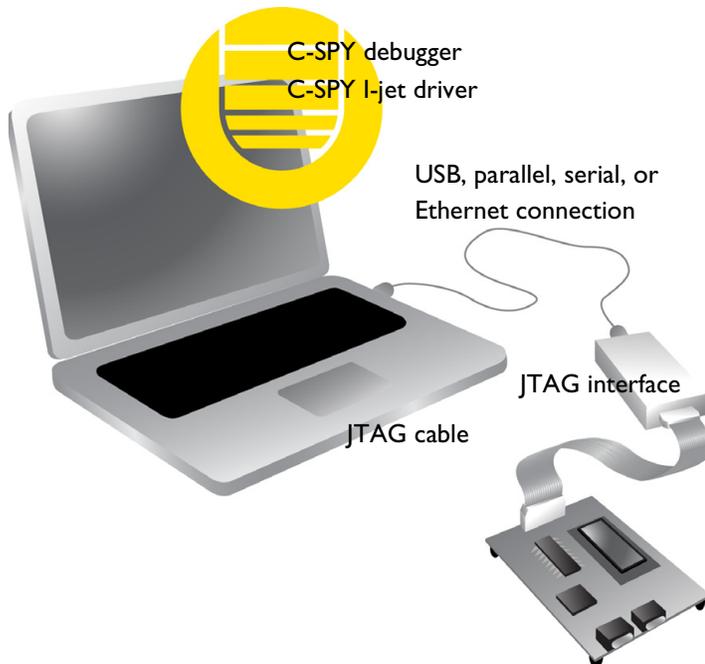
IAR Embedded Workbench for RISC-V comes with a C-SPY hardware debugger driver for the I-jet and I-jet Trace debug probes.

These topics are covered:

- Communication overview
- Hardware installation
- USB driver installation

COMMUNICATION OVERVIEW

The C-SPY hardware debugger driver uses USB to communicate with the hardware debugger. The hardware debugger communicates with the JTAG interface on the microcontroller.



When a 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 USB driver on the IAR Embedded Workbench installation media.

HARDWARE INSTALLATION

For best results, follow these steps.

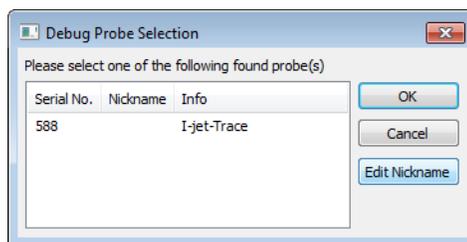
Recommended power-up sequence

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, debug probe, and C-SPY:

- 1 Connect the probe to the target board.
- 2 Connect the USB cable to the debug probe.
- 3 Power up the debug probe, if it is not powered via USB.
- 4 Power up the target board, if it is not powered via the debug probe.
- 5 Start the C-SPY debug session.
- 6 If more than one debug probe is connected to your computer, the **Debug Probe Selection** dialog box is displayed. In the dialog box, select the probe to use and click **OK**. For more information, see *--drv_communication*, page 345.

To give the probe a nickname, select the probe in the dialog box and click the **Edit Nickname** button. The nickname is saved locally on your computer and is available when opening other projects.

Note: The **Edit Nickname** button might not be available for the C-SPY driver you are using.



USB DRIVER INSTALLATION

C-SPY needs a USB driver, which for some probes, is automatically installed. If the USB driver is not installed automatically, you will need to install it manually.

Installing the I-jet USB driver

I-jet does not require any special driver software installation. Normally, all drivers for I-jet are automatically installed as part of the installation of IAR Embedded Workbench.

If you need to install the USB driver manually, navigate to `\Program Files\IAR Systems\Embedded Workbench x.x\riscv\drivers\ijet\USB\32-bit` or `64-bit` (depending on your system). Start the `dpinst.exe` application. This will install the USB driver. Note that the USB3 drivers are in the `\riscv\drivers\ijet\USB3` directory.

Installing the I-jet Trace USB driver

I-jet Trace does not require any special driver software installation. Normally, all drivers for I-jet Trace are automatically installed as part of the installation of IAR Embedded Workbench.

If you need to install the USB driver manually, navigate to `\riscv\drivers\jet\USB3\32-bit` or `64-bit` (in the installation directory). Start the `dpinst.exe` application. This will install the USB driver.

The USB LED will flash twice after enumerating on the USB2 ports, and three times on USB3 ports.

Getting started using C-SPY

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

Setting up C-SPY

These tasks are covered:

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

SETTING UP FOR DEBUGGING

- 1** Before you start C-SPY, choose **Project>Options>Debugger>Setup** and select the C-SPY driver that matches your debugger system—simulator or hardware debugger system.
- 2** In the **Category** list, select the appropriate C-SPY driver and make your settings. For information about these options, see *Debugger options*, page 373.
- 3** Click **OK**.
- 4** Choose **Tools>Options** to open the **IDE Options** dialog box:
 - Select **Debugger** to configure the debugger behavior
 - Select **Stack** to configure the debugger's tracking of stack usage.

For more information about these options, see the *IDE Project Management and Building Guide for RISC-V*. See also *Adapting for target hardware*, page 47.

EXECUTING FROM RESET

The **Run to** option—available on the **Debugger>Setup** page—specifies a location you want C-SPY to run to when you start a debug session as well as after each reset. C-SPY will place a temporary breakpoint at this location and all code up to this point is executed before stopping at the location. Note that this temporary breakpoint is removed when the debugger stops, regardless of how. If you stop the execution before the **Run to** location has been reached, the execution will not stop at that location when you start the execution again.

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 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 unlimited.

USING A SETUP MACRO FILE

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

For more information about setup macro files and functions, see *Introduction to C-SPY macros*, page 267.

For an example of how to use a setup macro file, see *Initializing target hardware before C-SPY starts*, page 48.

To register a setup macro file:

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

SELECTING A DEVICE DESCRIPTION FILE

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

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

For more information about device description files, see *Adapting for target hardware*, page 47.

To override the default device description file:

- 1 Before you start C-SPY, choose **Project>Options>Debugger>Setup**.
- 2 Select the **Override default** option, and choose a file using the **Device description file** browse button.

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

LOADING PLUGIN MODULES

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

For more information, see *Plugins*, page 380.

Starting C-SPY

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

These tasks are covered:

- Starting a debug session
- Loading executable files built outside of the IDE
- Starting a debug session with source files missing
- Loading multiple debug images
- Editing in C-SPY windows

STARTING A DEBUG SESSION

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



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



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

LOADING EXECUTABLE FILES BUILT OUTSIDE OF THE IDE

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

To create a project for an externally built file:

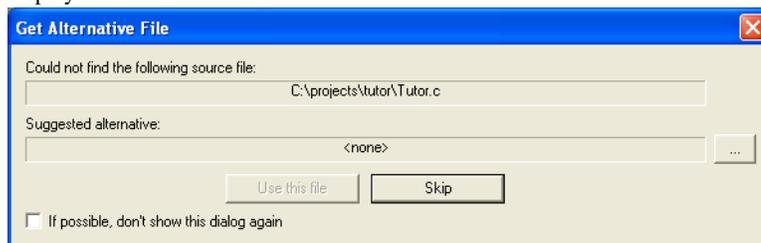
- 1 Choose **Project>Create New Project**, and specify a project name.
- 2 To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the file type drop-down list. Locate the executable file.
- 3 To start the executable file, click the **Download and Debug** button. The project can be reused whenever you rebuild your executable file.

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

STARTING A DEBUG SESSION WITH SOURCE FILES MISSING

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

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



Typically, you can use the dialog box like this:

- The source files are not available—Select **If possible, don't show this dialog again** and then click **Skip**. C-SPY will assume that there 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, select **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 selected **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 54.

LOADING MULTIPLE DEBUG IMAGES

Normally, a debuggable application consists of a single file that you debug. However, you can also load additional debug files (debug images). This means that the complete program consists of several debug 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 debug image has been loaded, you will have access to the combined debug information for all the loaded debug images. In the **Images** window you can choose whether you want to have access to debug information for a single debug image or for all images.

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

```
myPtr;3
```

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

```
myArray;5,10
```

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

```
myPtr;5,10
```

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

Adapting for target hardware

These tasks are covered:

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

MODIFYING A DEVICE DESCRIPTION FILE

C-SPY uses device description files provided with the product to handle several of the target-specific adaptations, see *Selecting a device description file*, page 43. Device description files contain device-specific information such as:

- Memory information for device-specific memory zones, see *C-SPY memory zones*, page 130. If you are using a C-SPY hardware debugger driver, the memory information retrieved from the device description file is not always sufficient, see *Memory Configuration dialog box for C-SPY hardware debugger drivers*, page 166.
- Definitions of memory-mapped peripheral units, device-specific CPU registers, and groups of these.

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

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



If you are using an I-jet or I-jet Trace debug probe, and the modified device description file contains modified memory ranges, make sure to select the option **Use Factory** in the **Memory Configuration** dialog box.

For information about how to load a device description file, see *Selecting a device description file*, page 43.

INITIALIZING TARGET HARDWARE BEFORE C-SPY STARTS

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

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

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

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

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

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

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

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

Reference information on starting C-SPY

Reference information about:

- *C-SPY Debugger main window*, page 49
- *Images window*, page 53
- *Get Alternative File dialog box*, page 54

See also:

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

C-SPY Debugger main window

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

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

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

Menu bar

These menus are available during a debug session:

Debug

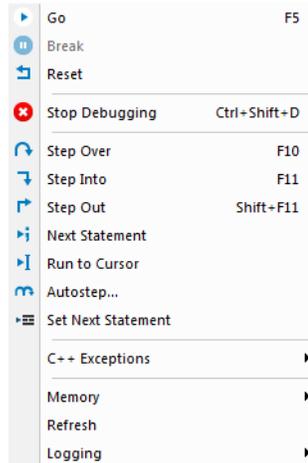
Provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

C-SPY driver menu

Provides commands specific to a C-SPY driver. The driver-specific menu is only available when the driver is used. For information about the driver-specific menu commands, see *Reference information on C-SPY driver menus*, page 391.

Debug menu

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



These commands are available:



Go (F5)

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

Note: If you are using symmetric multicore debugging, the **Go** command starts only the core in focus.



Break

Stops the application execution.

Note: If you are using symmetric multicore debugging, the **Break** command stops only the core in focus.



Reset

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

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

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



Stop Debugging (Ctrl+Shift+D)

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



Step Over (F10)

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



Step Into (F11)

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



Step Out (Shift+F11)

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



Next Statement

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



Run to Cursor

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



Autostep

Displays a dialog box where you can customize and perform autostepping, see *Autostep settings dialog box*, page 75.



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++ With exceptions**.

This menu command is not supported by your product package.

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++ With exceptions**.

This menu command is not supported by your product package.

Memory>Save

Displays a dialog box where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 142.

Memory>Restore

Displays a dialog box where you can load the contents of a file in, for example Intel-extended or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 143.

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.

Logging>Set Terminal I/O Log file

Displays a dialog box where you can choose to log simulated target access communication to a file. You can select the destination of the log file. See *Terminal I/O Log File dialog box*, page 73.

C-SPY windows

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

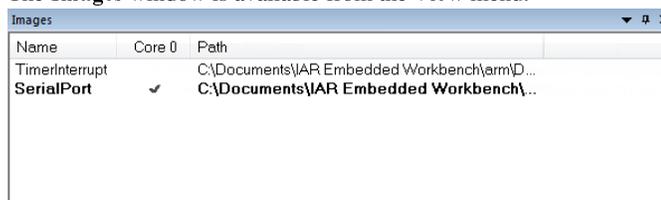
- C-SPY Debugger main window
- Disassembly window
- Memory window
- Symbolic Memory window
- Registers window
- Watch window
- Locals window
- Auto window
- Live Watch window

- Quick Watch window
- Statics window
- Call Stack window
- Trace window
- Function Trace window
- Timeline window, see *Reference information on application timeline*, page 201
- 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.

Images window

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



Name	Core 0	Path
TimerInterrupt		C:\Documents\IAR Embedded Workbench\arm\D...
SerialPort	✓	C:\Documents\IAR Embedded Workbench\...

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

Normally, a source application consists of a single debug image that you debug. However, you can also load additional images. This means that the complete debuggable unit consists of several debug images. See also *Loading multiple debug images*, page 45.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

C-SPY can use debug information from one or more of the loaded debug images simultaneously. Double-click on a row to make C-SPY use debug information from that debug image. The current choices are highlighted.

This area lists the loaded debug images in these columns:

Name

The name of the loaded debug image.

Core *N*

Double-click in this column to toggle using debug information from the debug image when that core is in focus.

Path

The path to the loaded debug image.

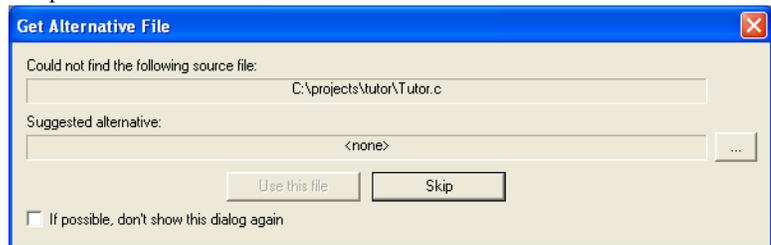
Related information

For related information, see:

- *Loading multiple debug images*, page 45
- *Images*, page 376
- *__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.



See also *Starting a debug session with source files missing*, page 44.

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 44.

Executing your application

- Introduction to application execution
- Reference information on application execution

Introduction to application execution

These topics are covered:

- Briefly about application execution
- Source and disassembly mode debugging
- Single stepping
- Troubleshooting slow stepping speed
- Running the application
- Highlighting
- Viewing the call stack
- Terminal input and output
- Debug logging

BRIEFLY ABOUT APPLICATION EXECUTION

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

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

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

SOURCE AND DISASSEMBLY MODE DEBUGGING

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

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

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

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

SINGLE STEPPING

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

There are several factors that can slow down the stepping speed. If you find it too slow, see *Troubleshooting slow stepping speed*, page 60 for some tips.

The step commands

There are four step commands:

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

Using the **Autostep settings** dialog box, you can automate the single stepping. For more information, see *Autostep settings dialog box*, page 75.

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.

TROUBLESHOOTING SLOW STEPPING SPEED

If you find that stepping speed is slow, these troubleshooting tips might speed up stepping:

- If you are using a hardware debugger system, keep track of how many hardware breakpoints that are used and make sure some of them are left for stepping.

Stepping in C-SPY is normally performed using breakpoints. When C-SPY performs a step command, a breakpoint is set on the next statement and the application executes until it reaches this breakpoint. If you are using a hardware debugger system, the number of hardware breakpoints—typically used for setting a stepping breakpoint in code that is located in flash/ROM memory—is limited. If you, for example, step into a `C switch` statement, breakpoints are set on each branch; this might consume several hardware breakpoints. If the number of available hardware breakpoints is exceeded, C-SPY switches into single stepping on assembly level, which can be very slow.

For more information, see *Breakpoints in the C-SPY hardware debugger drivers*, page 108 and *Breakpoint consumers*, page 109.

- Disable trace data collection, using the **Enable/Disable** button in both the **Trace** and the **Function Profiling** windows. Trace data collection might slow down stepping because the collected trace data is processed after each step. Note that it is not sufficient to just close the corresponding windows to disable trace data collection.
- Choose to view only a limited selection of SFR registers. You can choose between two alternatives. Either type `#SFR_name` (where `SFR_name` reflects the name of the SFR you want to monitor) in the **Watch** window, or create your own filter for displaying a limited group of SFRs in the **Registers** window. Displaying many SFR registers might slow down stepping because all registers must be read from the hardware after each step. See *Defining application-specific register groups*, page 133.
- Close the **Memory** and **Symbolic Memory** windows if they are open, because the visible memory must be read after each step and that might slow down stepping.
- Close any window that displays expressions such as **Watch**, **Live Watch**, **Locals**, **Statics** if it is open, because all these windows read memory after each step and that might slow down stepping.
- Close the **Stack** window if it is open. Choose **Tools>Options>Stack** and disable the **Enable graphical stack display and stack usage tracking** option if it is enabled.
- If possible, increase the communication speed between C-SPY and the target board/emulator.

RUNNING THE APPLICATION



Go

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

Note: If you are using symmetric multicore debugging, the **Go** command starts only the core in focus.



Run to Cursor

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

HIGHLIGHTING

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

```

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

```

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

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

Code coverage

From the context menu in the **Code Coverage** window, you can toggle highlight colors and icons in the editor window that show code coverage analysis for the source code, see *Code Coverage window*, page 227.

These are the colors and icons that are used:

- Red highlight color and a red diamond—the code range has not been executed.
- Green highlight color—100% of the code range has been executed.
- Yellow highlight color and a red diamond—parts of the code range have been executed.

This figure illustrates all three code coverage highlight colors:

```

callCount = 0;
InitFib();
while (callCount < MAX_FIB)
{
    DoForegroundProcess();
}

```

VIEWING THE CALL STACK

The compiler generates extensive call frame 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 call frame 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 more information, see the *IAR Assembler User Guide for RISC-V*.

Note: For highly optimized code, C-SPY might not be able to identify all calls. This means that for highly optimized code, the call stack is not entirely trustworthy.

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 72 and *Terminal I/O Log File dialog box*, page 73.

DEBUG LOGGING

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



It can sometimes be convenient to log the information to a file where you can easily inspect it, see *Debug Log window*, page 73. The two main advantages are:

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

Reference information on application execution

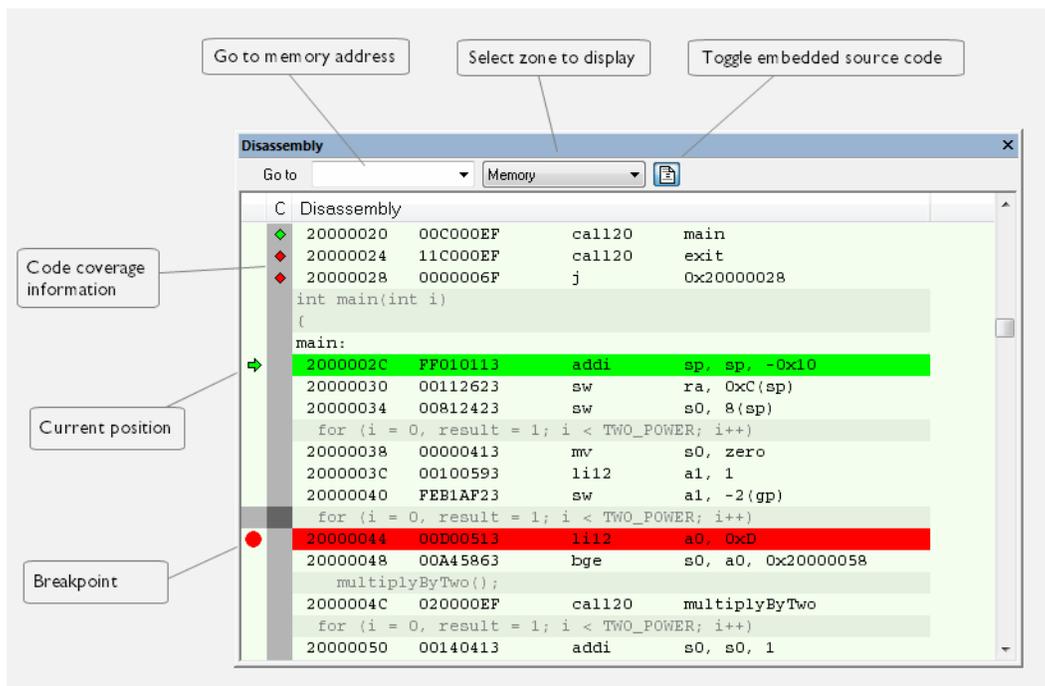
Reference information about:

- *Disassembly window*, page 65
- *Call Stack window*, page 70
- *Terminal I/O window*, page 72
- *Terminal I/O Log File dialog box*, page 73
- *Debug Log window*, page 73
- *Report Assert dialog box*, page 75
- *Autostep settings dialog box*, page 75

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

Disassembly window

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



This figure reflects the C-SPY simulator.

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 color 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.

See also *Source and disassembly mode debugging*, page 57.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

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:

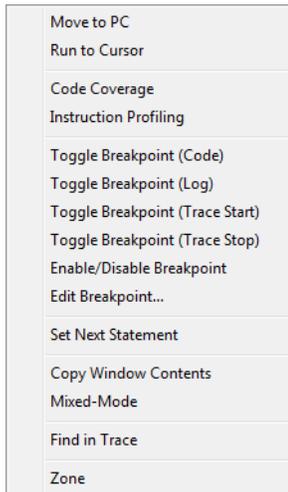
Green highlight color	Indicates the current position, that is the next assembler instruction to be executed. To move the cursor to any line in the Disassembly window, click the line. Alternatively, move the cursor using the navigation keys.
Yellow highlight color	Indicates a position other than the current 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>Breakpoints</i> , page 105.
Green diamond	Code coverage icon—indicates code that has been executed.
Red diamond	Code coverage icon—indicates code that has <i>not</i> been executed.
Red/yellow diamond (red top/yellow bottom)	Code coverage icon—indicates a branch that is <i>never</i> taken.
Red/yellow diamond (red left side/yellow right side)	Code coverage icon—indicates a branch that is <i>always</i> taken.

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. For debug probes that support it, C-SPY can capture full instruction trace in real time.

Context menu

This context menu is available:



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

These commands are available:

Move to PC

Displays code at the current program counter location.

Run to Cursor

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

Code Coverage

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

Enable

Toggles code coverage on or off.

Show

Toggles the display of code coverage on or off. Code coverage is indicated by a red, green, and red/yellow diamonds in the left margin.

Clear	Clears all code coverage information.
Next Different Coverage >	Moves the insertion point to the next line in the window with a different code coverage status than the selected line.
Previous Different Coverage <	Moves the insertion point to the closest preceding line in the window with a different code coverage status than the selected line.

Instruction Profiling

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

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

Toggle Breakpoint (Code)

Toggles a code breakpoint. Assembler instructions and any corresponding label at which code breakpoints have been set are highlighted in red. For more information, see *Code breakpoints dialog box*, page 118.

Toggle Breakpoint (Log)

Toggles a log breakpoint for trace printouts. Assembler instructions at which log breakpoints have been set are highlighted in red. For more information, see *Log breakpoints dialog box*, page 120.

Toggle Breakpoint (Trace Start)

Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Start Trigger breakpoint dialog box*, page 191.

Toggle Breakpoint (Trace Stop)

Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Stop Trigger breakpoint dialog box*, page 193.

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.

Find in Trace

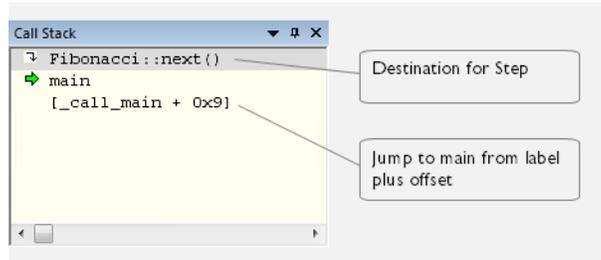
Searches the contents of the **Trace** window for occurrences of the given location—the position of the insertion point in the source code—and reports the result in the **Find in Trace** window. This menu command requires support for Trace in the C-SPY driver you are using, see *Differences between the C-SPY drivers*, page 35.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Call Stack window

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



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

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

See also *Viewing the call stack*, page 63.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

Each entry in the display area is formatted in one of these ways:

`function(values)***`

A C/C++ function with debug information.

Provided that **Show Arguments** is enabled, *values* is a list of the current values of the parameters, or empty if the function does not take any parameters.

***, if present, indicates that the function has been inlined by the compiler. For information about function inlining, see the *IAR C/C++ Development Guide for RISC-V*.

`[label + offset]`

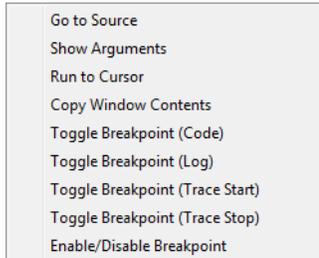
An assembler function, or a C/C++ function without debug information.

`<exception_frame>`

An interrupt.

Context menu

This context menu is available:



These commands are available:

Go to Source

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

Show Arguments

Shows function arguments.

Run to Cursor

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

Copy Window Contents

Copies the contents of the **Call Stack** window and stores them on the clipboard.

Toggle Breakpoint (Code)

Toggles a code breakpoint.

Toggle Breakpoint (Log)

Toggles a log breakpoint.

Toggle Breakpoint (Trace Start)

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

Toggle Breakpoint (Trace Stop)

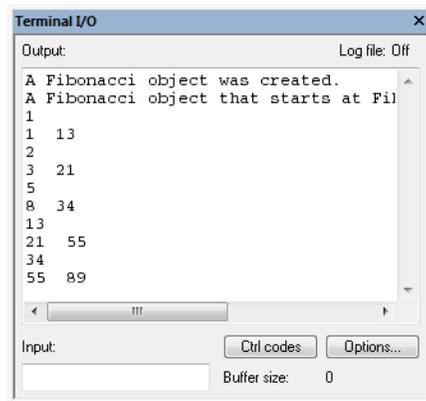
Toggles a Trace Stop breakpoint. When the breakpoint is triggered, trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports it.

Enable/Disable Breakpoint

Enables or disables the selected breakpoint.

Terminal I/O window

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



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

To use this window, you must:

- I Link your application with the option **Include C-SPY debugging support**.

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.

See also *Terminal input and output*, page 63.

Requirements

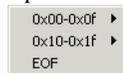
Can be used with all C-SPY debugger drivers and debug probes.

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.

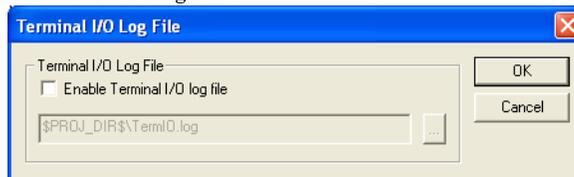


Options

Opens the **IDE Options** dialog box where you can set options for terminal I/O. For information about the options available in this dialog box, see *Terminal I/O options in IDE Project Management and Building Guide for RISC-V*.

Terminal I/O Log File dialog box

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



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

See also *Terminal input and output*, page 63.

Requirements

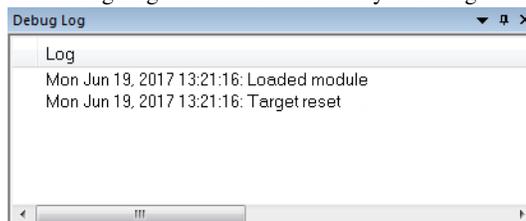
Can be used with all C-SPY debugger drivers and debug probes.

Terminal I/O Log File

Controls the logging of terminal I/O. To enable logging of terminal I/O to a file, select **Enable Terminal I/O 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>Debug Log**.



This window displays debugger output, such as diagnostic messages, macro-generated output, and information about trace. When opened, this window is, by default, grouped together with the other message windows, see *IDE Project Management and Building Guide for RISC-V*.

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>
```

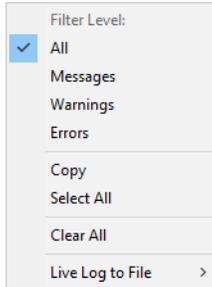
See also *Debug logging*, page 64.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Context menu

This context menu is available:



These commands are available:

All

Shows all messages sent by the debugging tools and drivers.

Messages

Shows all C-SPY messages.

Warnings

Shows warnings and errors.

Errors

Shows errors only.

Copy

Copies the contents of the window.

Select All

Selects the contents of the window.

Clear All

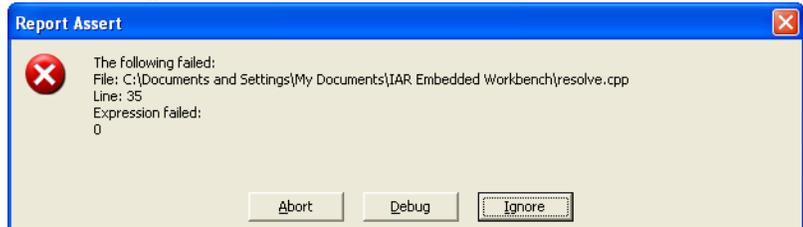
Clears the contents of the window.

Live Log to File

Displays a submenu with commands for writing the debug messages to a log file and setting filter levels for the log.

Report Assert dialog box

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



Abort

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

Debug

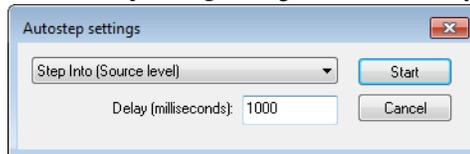
C-SPY stops the execution of the application and returns control to you.

Ignore

The assertion is ignored and the application continues to execute.

Autostep settings dialog box

The **Autostep settings** dialog box is available by choosing **Debug>Autostep**.



Use this dialog box to configure autostepping.

Select the step command you want to automate from the drop-down menu. The step will be performed with the specified interval. For a description of the available step commands, see *Single stepping*, page 58. You can stop the autostepping by clicking the Break button on the debug toolbar.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Delay (milliseconds)

The delay between each step command in milliseconds. The step is repeated with this interval.

Variables and expressions

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

Introduction to working with variables and expressions

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

These topics are covered:

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

BRIEFLY ABOUT WORKING WITH VARIABLES AND EXPRESSIONS

There are several methods for looking at variables and calculating their values. These methods are suitable for basic debugging:

- **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 **Macro Quicklaunch** window and the **Quick Watch** window give you precise control over when to evaluate an expression.
- The **Symbols** window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.

These additional methods for looking at variables are suitable for more advanced analysis:

- The **Data Log** window and the **Data Log Summary** window display logs of accesses to up to four different memory locations 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.

For more information about these windows, see *The application timeline*, page 197.

C-SPY EXPRESSIONS

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

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

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

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

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

C/C++ symbols

C symbols are symbols that you have defined in the C source code of your application, for instance variables, constants, and functions (functions can be used as symbols but cannot be executed). C symbols can be referenced by their names. Note that C++

symbols might implicitly contain function calls which are not allowed in C-SPY symbols and expressions.

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

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

However, this line will be accepted:

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

Assembler symbols

Assembler symbols can be assembler labels or registers, for example the program counter, the stack pointer, or other CPU registers. If a device description file is used, all memory-mapped peripheral units, such as I/O ports, can also be used as assembler symbols in the same way as the CPU registers. See *Modifying a device description file*, page 47.

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

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

Table 4: C-SPY assembler symbols expressions

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

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

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

Which processor-specific symbols are available by default can be seen in the **Registers** window, using the `CPU Registers` register group. See *Registers window*, page 152.

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 268.

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 274.

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 use the lowest optimization level during compilation, that is, **None**.

Working with variables and expressions

These tasks are covered:

- Using the windows related to variables and expressions
- Viewing assembler variables

See also *Analyzing your application's timeline*, page 198.

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 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**, **Live Watch**, and **Quick Watch** windows, you can select a different interpretation to better suit the declaration of the variables.

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

The screenshot shows the C-SPY interface with the assembler source file on the left and the Watch window on the right. The source file contains the following code:

```

NAME main

PUBLIC __iar_program_start

SECTION .intvec : CODE (2)
CODE32

__iar_program_start
B main

SECTION .text : CODE (2)

asmvar1: DC32 42
asmvar2: DC32 456
asmvar3: DC8 55
asmvar4: DC8 10

CODE32

main NOB
B main

END

```

The Watch window displays the following data:

Expression	Value	Location	Type
asmvar1	42	0x00000080	int
asmvar2	456	0x00000084	int
asmvar3	55	0x00000088	<8-bit unsigned>
asmvar4	2615	0x0000008c	int

The context menu for the `asmvar4` row shows the following options:

- Default Format
- Binary Format
- Octal Format
- Decimal Format
- Hexadecimal Format
- Char Format
- Show As
 - As Is
 - 8-bit Signed
 - 8-bit Unsigned**
 - 16-bit Signed
 - 16-bit Unsigned
 - 32-bit Signed
 - 32-bit Unsigned
 - 64-bit Signed
 - 64-bit Unsigned
 - float
 - double
- Save to File...

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

Reference information on working with variables and expressions

Reference information about:

- *Auto window*, page 83
- *Locals window*, page 86
- *Watch window*, page 88
- *Live Watch window*, page 91
- *Statics window*, page 94
- *Quick Watch window*, page 97
- *Symbols window*, page 100
- *Resolve Symbol Ambiguity dialog box*, page 103

See also:

- *Reference information on trace*, page 179 for trace-related reference information
- *Macro Quicklaunch window*, page 331

Auto window

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

Expression	Value	Location	Type
NextCounter	NextCounter (0x40B)		void (__...
fib	1	Memory: 0xFEF74	uint32_t
GetFib	GetFib (0x141)		uint32_t(...
callCount	3	Memory: 0xFEFA8	signed int

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

See also *Editing in C-SPY windows*, page 46.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

This area contains these columns:

Expression

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

Value

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

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

This column is editable.

Location

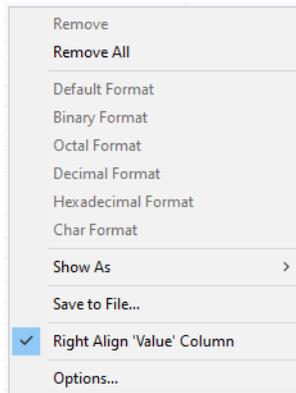
The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:



Note: The contents of this menu are dynamic and depend on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

Default Format

Binary Format

Octal Format

Decimal Format

Hexadecimal Format

Char Format

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 82.

Save to File

Saves content to a file in a tab-separated format.

Right Align 'Value' Column

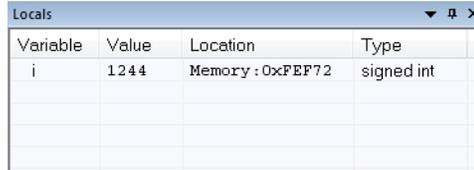
Right-aligns the contents of the **Value** column.

Options

Displays the **IDE Options** dialog box where you can set various options for C-SPY windows.

Locals window

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



Variable	Value	Location	Type
i	1244	Memory : 0xPEF72	signed int

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

See also *Editing in C-SPY windows*, page 46.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

This area contains these columns:

Variable

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

Value

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

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

This column is editable.

Location

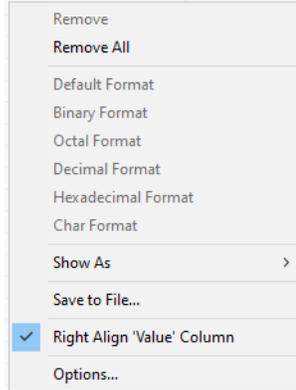
The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:



Note: The contents of this menu are dynamic and depend on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

Default Format

Binary Format

Octal Format

Decimal Format

Hexadecimal Format

Char Format

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

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

Variables

The display setting affects only the selected variable, not other variables.

Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 82.

Save to File

Saves content to a file in a tab-separated format.

Right Align 'Value' Column

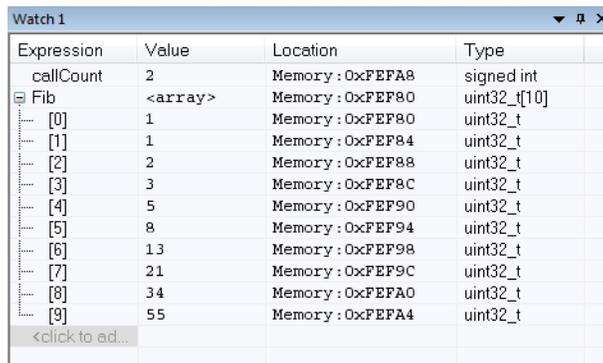
Right-aligns the contents of the **Value** column.

Options

Displays the **IDE Options** dialog box where you can set various options for C-SPY windows.

Watch window

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



Expression	Value	Location	Type
callCount	2	Memory : 0xFEFA8	signed int
[-] Fib	<array>	Memory : 0xFEFA0	uint32_t[10]
[0]	1	Memory : 0xFEFA0	uint32_t
[1]	1	Memory : 0xFEFA4	uint32_t
[2]	2	Memory : 0xFEFA8	uint32_t
[3]	3	Memory : 0xFEFA8C	uint32_t
[4]	5	Memory : 0xFEFA90	uint32_t
[5]	8	Memory : 0xFEFA94	uint32_t
[6]	13	Memory : 0xFEFA98	uint32_t
[7]	21	Memory : 0xFEFA9C	uint32_t
[8]	34	Memory : 0xFEFAA0	uint32_t
[9]	55	Memory : 0xFEFAA4	uint32_t
<click to ad...			

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

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



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

See also *Editing in C-SPY windows*, page 46.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

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.

Value

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

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

This column is editable.

Location

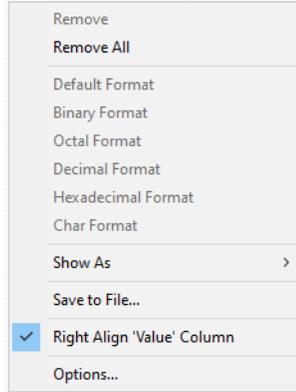
The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:



Note: The contents of this menu are dynamic and depend on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

Default Format

Binary Format

Octal Format

Decimal Format

Hexadecimal Format

Char Format

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

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

Variables

The display setting affects only the selected variable, not other variables.

Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration type—are affected by the display setting.

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 82.

Save to File

Saves content to a file in a tab-separated format.

Right Align 'Value' Column

Right-aligns the contents of the **Value** column.

Options

Displays the **IDE Options** dialog box where you can set various options for C-SPY windows.

Live Watch window

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

Expression	Value	Location	Type
GetFib	GetFib (0x141)		uint32_t(__ne...
...	GetFib (0x141)	Memory: 0x141	uint32_t(int_f...
<click to ad...			

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.

See also *Editing in C-SPY windows*, page 46.

Requirements

One of these alternatives:

- The C-SPY simulator

- The I-jet driver and a device that supports runtime memory access via the system bus

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.

Value

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

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

This column is editable.

Location

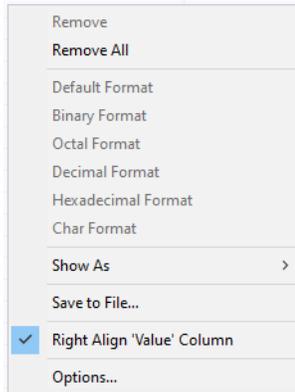
The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:



Note: The contents of this menu are dynamic and depend on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

Default Format

Binary Format

Octal Format

Decimal Format

Hexadecimal Format

Char Format

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 82.

Save to File

Saves content to a file in a tab-separated format.

Right Align 'Value' Column

Right-aligns the contents of the **Value** column.

Options

Displays the **IDE Options** dialog box where you can set various options for C-SPY windows.

Statics window

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

Variable	Value	Location	Type	Module
fibStat <UsingClasses\FibStat>	<class>	Memory : 0xFB140	class Fibonacci	UsingClasses
mCurrent	2	Memory : 0xFB140	uint_fast8_t	
msFib <FibonacciByClass\Fibonacci::msFib>	size=100	Memory : 0xFB134	class vector<uint32_t>	FibonacciByClass
<Raw>	<class>	Memory : 0xFB134	class vector<uint32_t>	
MyImpl	<class>	Memory : 0xFB134	vector<uint32_t>::Impl	
<class>	<class>	Memory : 0xFB134	class _Vector_value<allocator...	
<class>	<class>	Memory : 0xFB134	class _ClassUtil::_AllocHolder...	
_Myfirst	0xA0E0	Memory : 0xFB134	void __near*	
_Mylast	0xA270	Memory : 0xFB136	void __near*	
_Myend	0xA270	Memory : 0xFB138	void __near*	
<0>	0	Memory : 0xFA0E0	uint32_t	
<1>	0	Memory : 0xFA0E4	uint32_t	

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

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

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

See also *Editing in C-SPY windows*, page 46.

To select variables to monitor:

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

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

This area contains these columns:

Variable

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

Value

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

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

This column is editable.

Location

The location in memory where this variable is stored.

Type

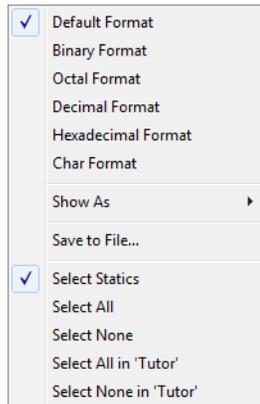
The data type of the variable.

Module

The module of the variable.

Context menu

This context menu is available:



These commands are available:

Default Format

Binary Format

Octal Format

Decimal Format

Hexadecimal Format

Char Format

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 82.

Save to File

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

Select Statics

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

Select All

Selects all variables.

Select None

Deselects all variables.

Select All in *module*

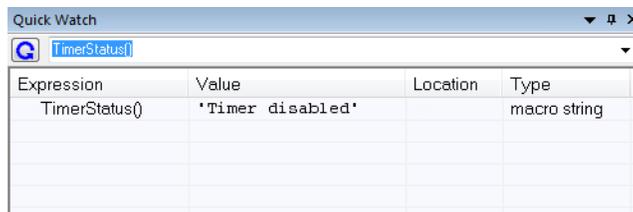
Selects all variables in the selected module.

Select None in *module*

Deselects all variables in the selected module.

Quick Watch window

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



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

In contrast to the **Watch** window, the **Quick Watch** window gives you precise control over when to evaluate the expression. For single variables this might not be necessary,

but for expressions with possible side effects, such as assignments and C-SPY macro functions, it allows you to perform evaluations under controlled conditions.

See also *Editing in C-SPY windows*, page 46.

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:

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



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

For an example, see *Using C-SPY macros*, page 269.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

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.

Value

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

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

This column is editable.

Location

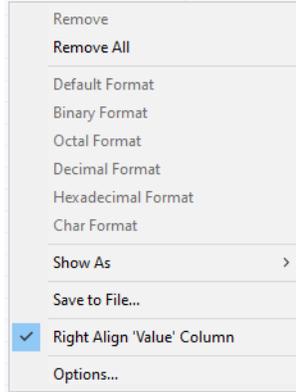
The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:



Note: The contents of this menu are dynamic and depend on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

Default Format

Binary Format

Octal Format

Decimal Format

Hexadecimal Format

Char Format

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

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

Variables

The display setting affects only the selected variable, not other variables.

Array elements The display setting affects the complete array, that is, the same display format is used for each array element.

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 82.

Save to File

Saves content to a file in a tab-separated format.

Right Align 'Value' Column

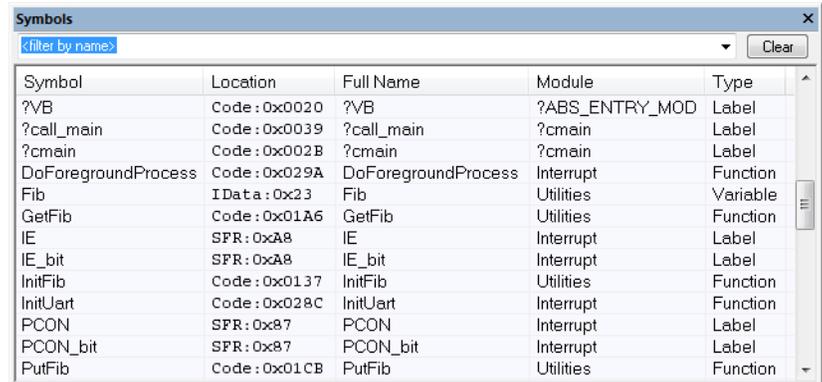
Right-aligns the contents of the **Value** column.

Options

Displays the **IDE Options** dialog box where you can set various options for C-SPY windows.

Symbols window

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



Symbol	Location	Full Name	Module	Type
?VB	Code : 0x0020	?VB	?ABS_ENTRY_MOD	Label
?call_main	Code : 0x0039	?call_main	?cmain	Label
?cmain	Code : 0x002B	?cmain	?cmain	Label
DoForegroundProcess	Code : 0x029A	DoForegroundProcess	Interrupt	Function
Fib	IData : 0x23	Fib	Utilities	Variable
GetFib	Code : 0x01A6	GetFib	Utilities	Function
IE	SFR : 0xA8	IE	Interrupt	Label
IE_bit	SFR : 0xA8	IE_bit	Interrupt	Label
InitFib	Code : 0x0137	InitFib	Utilities	Function
InitUart	Code : 0x028C	InitUart	Interrupt	Function
PCON	SFR : 0x87	PCON	Interrupt	Label
PCON_bit	SFR : 0x87	PCON_bit	Interrupt	Label
PutFib	Code : 0x01CB	PutFib	Utilities	Function

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.

You can drag the contents of cells in the **Symbol**, **Location**, and **Full Name** columns and drop in some other windows in the IDE.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Toolbar

The toolbar contains:

<filter by name>

Type the first characters of the symbol names that you want to find, and press Enter. All symbols (of the types you have selected on the context menu) whose name starts with these characters will be displayed. If you have chosen not to display some types of symbols, the window will list how many of those that were found but are not displayed.

Use the drop-down list to use old search strings. The search box has a history depth of eight search entries.

Clear

Cancels the effects of the search filter and restores all symbols in the window.

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.

Module

The program module where the symbol is defined.

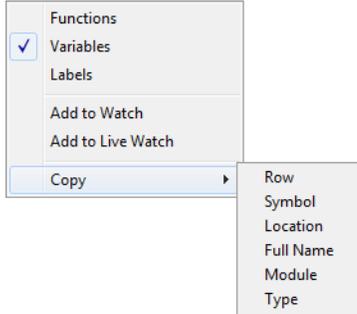
Type

The symbol type, whether it is a function, label, or variable.

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

Context menu

This context menu is available:



These commands are available:

Functions

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

Variables

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

Labels

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

Add to Watch

Adds the selected symbol to the **Watch** window.

Add to Live Watch

Adds the selected symbol to the **Live Watch** window.

Copy

Copies the contents of the cells on the selected line.

Row Copies all contents of the selected line.

Symbol Copies the contents of the **Symbol** cell on the selected line.

Location Copies the contents of the **Location** cell on the selected line.

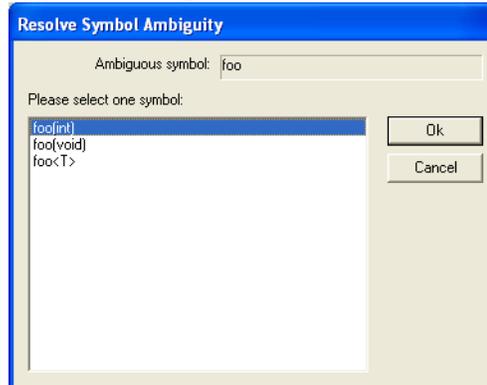
Full Name Copies the contents of the **Full Name** cell on the selected line.

Module Copies the contents of the **Module** cell on the selected line.

Type Copies the contents of the **Type** cell on the selected line.

Resolve Symbol Ambiguity dialog box

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



Requirements

Can be used with all C-SPY debugger drivers and debug probes.

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.

Breakpoints

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

Introduction to setting and using breakpoints

These topics are covered:

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

REASONS FOR USING BREAKPOINTS

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

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

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

BRIEFLY ABOUT SETTING BREAKPOINTS

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

appear in the **Breakpoints** window. From this window you can conveniently view all breakpoints, enable and disable breakpoints, and open a dialog box for defining new breakpoints. The **Breakpoint Usage** window also lists all internally used breakpoints, see *Breakpoint consumers*, page 109.

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 58.

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

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

BREAKPOINT TYPES

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

Code breakpoints

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

Log breakpoints

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

Trace Start/Stop Trigger breakpoints

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

Data breakpoints

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

small parts of the code. Data breakpoints are triggered when data is accessed at the specified location.

The execution will usually stop directly after the instruction that accessed the data has been executed.

Data Log breakpoints

Data log breakpoints are triggered when a specified memory address is accessed. A log entry is written in the **Data Log** window for each access.

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

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

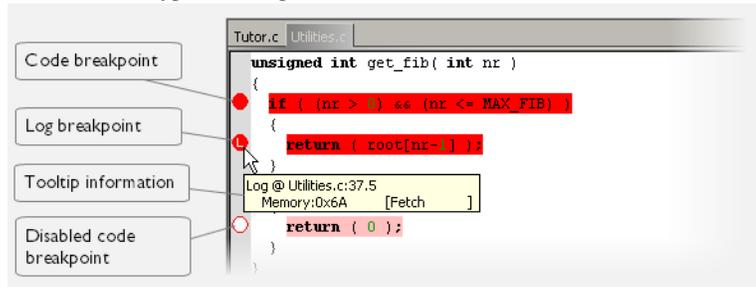
Immediate breakpoints

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

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

BREAKPOINT ICONS

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



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



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

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

BREAKPOINTS IN THE C-SPY SIMULATOR

The C-SPY simulator supports all breakpoint types. The number of breakpoints is unlimited.

BREAKPOINTS IN THE C-SPY HARDWARE DEBUGGER DRIVERS

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

If there are software breakpoints available, the debugger will first use these before using hardware breakpoints. Exceeding the number of available hardware breakpoints, when software breakpoints are not available, 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** window and in the **Breakpoints** window, for example **Data @[R] callCount**.

C-SPY itself

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

- The debugger option **Run to** has been selected, and any step command is used. These are temporary breakpoints which are only set during a debug session. This means that they are not visible in the **Breakpoints** window.
- The linker option **Include C-SPY debugging support** has been selected. In the DLIB runtime environment, C-SPY will set a system breakpoint on the `__DebugBreak` label.

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

C-SPY plugin modules

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

To disable the breakpoint used by the Stack window:

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

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

Setting breakpoints

These tasks are covered:

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

VARIOUS WAYS TO SET A BREAKPOINT

You can set a breakpoint in various ways:

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

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

TOGGLING A SIMPLE CODE BREAKPOINT

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:

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



SETTING BREAKPOINTS USING THE DIALOG BOX

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

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

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

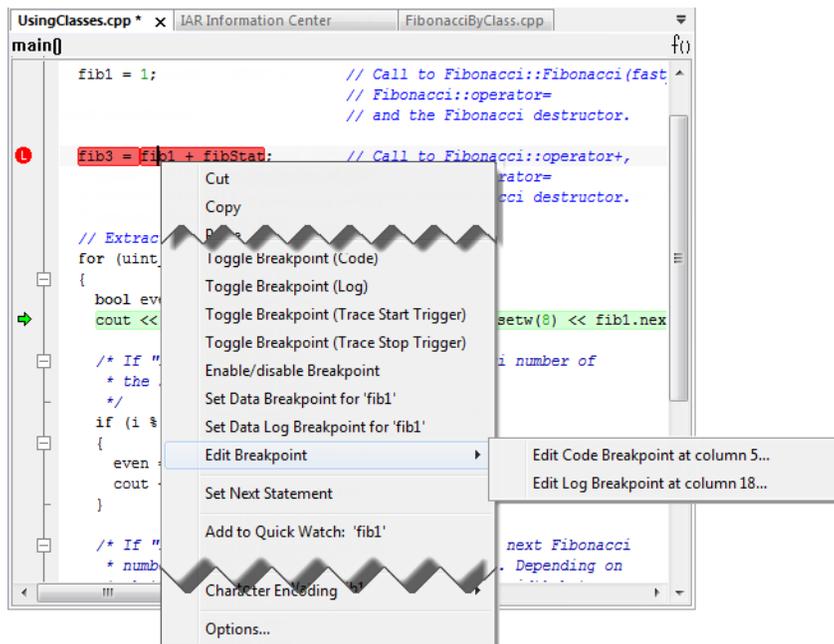
To set a new breakpoint:

- 1 Choose **View>Breakpoints** to open the **Breakpoints** window.
- 2 In the **Breakpoints** window, right-click, and choose **New Breakpoint** from the context menu.
- 3 On the submenu, choose the breakpoint type you want to set. Depending on the C-SPY driver you are using, different breakpoint types are available.
- 4 In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.

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

To modify an existing breakpoint:

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



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

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

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

SETTING A DATA BREAKPOINT IN THE MEMORY WINDOW

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

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

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

SETTING BREAKPOINTS USING SYSTEM MACROS

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

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

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

These breakpoint macros are available:

C-SPY macro for breakpoints	Simulator	I-jet	GDB Server
<code>__setCodeBreak</code>	Yes	Yes	Yes
<code>__setDataBreak</code>	Yes	Yes	Yes
<code>__setLogBreak</code>	Yes	Yes	—
<code>__setDataLogBreak</code>	Yes	—	—
<code>__setSimBreak</code>	Yes	—	—
<code>__setTraceStartBreak</code>	Yes	Yes	—
<code>__setTraceStopBreak</code>	Yes	Yes	—
<code>__clearBreak</code>	Yes	Yes	Yes

Table 6: C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 283.

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

You can use a setup macro file to define breakpoints at C-SPY startup. Follow the procedure described in *Using C-SPY macros*, page 269.

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

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

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

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



Performing a task and continuing execution

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

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

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

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

```
__var my_counter;

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

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

Reference information on breakpoints

Reference information about:

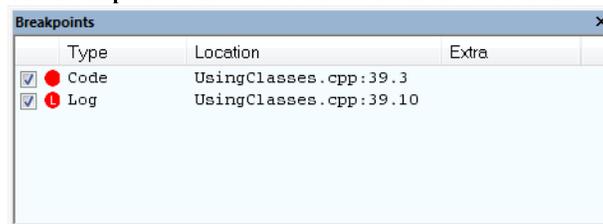
- *Breakpoints window*, page 115
- *Breakpoint Usage window*, page 117
- *Code breakpoints dialog box*, page 118
- *Log breakpoints dialog box*, page 120
- *Data breakpoints dialog box (Simulator)*, page 121
- *Data breakpoints dialog box (for C-SPY hardware debugger drivers)*, page 123
- *Data Log breakpoints dialog box*, page 124
- *Immediate breakpoints dialog box*, page 125
- *Enter Location dialog box*, page 126
- *Resolve Source Ambiguity dialog box*, page 128

See also:

- *Reference information on C-SPY system macros*, page 283
- *Reference information on trace*, page 179

Breakpoints window

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



This window lists all breakpoints you define.

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

Requirements

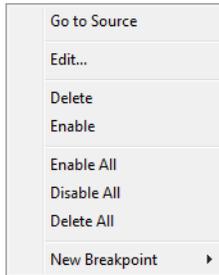
Can be used with all C-SPY debugger drivers and debug probes.

Display area

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

Context menu

This context menu is available:



These commands are available:

Go to Source

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

Edit

Opens the breakpoint dialog box for the breakpoint you selected.

Delete

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

Enable

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

Disable

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

Enable All

Enables all defined breakpoints.

Disable All

Disables all defined breakpoints.

Delete All

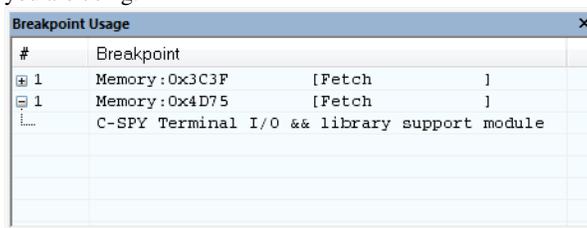
Deletes all defined breakpoints.

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	Memory:0x3C3F	[Fetch]
1	Memory:0x4D75	[Fetch]
	C-SPY Terminal I/O && library support module	

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

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

C-SPY uses breakpoints when stepping. If your target system has a limited number of hardware breakpoints, 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.

For more information, see *Breakpoints in the C-SPY hardware debugger drivers*, page 108.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

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

Code breakpoints dialog box

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

This figure reflects the C-SPY simulator.

Use the **Code** breakpoints dialog box to set a code breakpoint, see *Setting breakpoints using the dialog box*, page 110.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Break At

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

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.

Note: This option might not be supported by the combination of C-SPY driver and device that you are using.

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 113.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 78.

Condition true

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

Condition changed

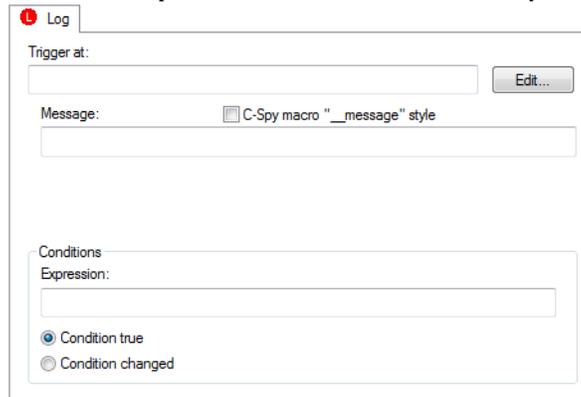
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Skip count

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

Log breakpoints dialog box

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



This figure reflects the C-SPY simulator.

Use the **Log** breakpoints dialog box to set a log breakpoint, see *Setting breakpoints using the dialog box*, page 110.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

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 277.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 78.

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 (Simulator)

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

This figure reflects the C-SPY simulator.

Use the **Data** breakpoints dialog box to set a data breakpoint, see *Setting breakpoints using the dialog box*, page 110. Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed.

Requirements

The C-SPY simulator.

Break At

Specify the data location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read/Write

Reads from or writes to location.

Read

Reads from location.

Write

Writes to location.

Size

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

Auto

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

Manual

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

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

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 113.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 78.

Condition true

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

Condition changed

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

Skip count

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

Data breakpoints dialog box (for C-SPY hardware debugger drivers)

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.

Use the **Data** breakpoints dialog box to set a data breakpoint, see *Setting breakpoints using the dialog box*, page 110. Data breakpoints never stop execution within a single instruction. They are recorded and reported after the instruction is executed.

Requirements

Any C-SPY hardware debugger driver.

Break At

Specify the data location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read/Write

Reads from or writes to location.

Read

Reads from location.

Write

Writes to location.

Size

These options are not available in the current version of the product.

Match data

These options are not available in the current version of the product.

Trigger range

These options are not available in the current version of the product.

Data Log breakpoints dialog box

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



Use the **Data Log** breakpoints dialog box to set a maximum of four data log breakpoints on memory addresses, see *Setting breakpoints using the dialog box*, page 110.

See also *Data Log breakpoints*, page 107 and *Getting started using data logging*, page 201.

Requirements

The C-SPY simulator.

Break At

Specify a memory location as a variable (with static storage duration) or as an address.

Access Type

Selects the type of access to the variable that generates a log entry:

Read/Write

Read and write accesses from or writes to location of the variable.

Read

Read accesses from the location of the variable.

Write

Write accesses to location of the variable.

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.

In the C-SPY simulator, you can use the **Immediate** breakpoints dialog box to set an immediate breakpoint, see *Setting breakpoints using the dialog box*, page 110. Immediate breakpoints do not stop execution at all—they only suspend it temporarily.

Requirements

The C-SPY simulator.

Trigger at

Specify the data location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read

Reads from location.

Write

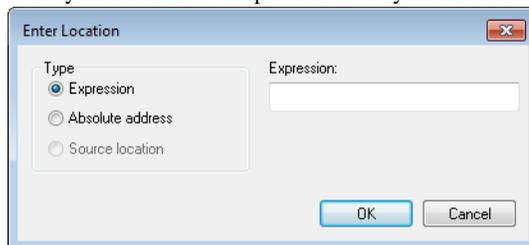
Writes to location.

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 113.

Enter Location dialog box

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



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

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

Type

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

Expression

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

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

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

For more information about C-SPY expressions, see *C-SPY expressions*, page 78.

Absolute address

An absolute location on the form *zone:hexaddress* or simply *hexaddress* (for example `Memory:0x42`). *zone* refers to C-SPY memory zones and specifies in which memory the address belongs, see *C-SPY memory zones*, page 130.

Source location

A location in your C source code using the syntax:

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

filename specifies the filename and full path.

row specifies the row in which you want the breakpoint.

column specifies the column in which you want the breakpoint.

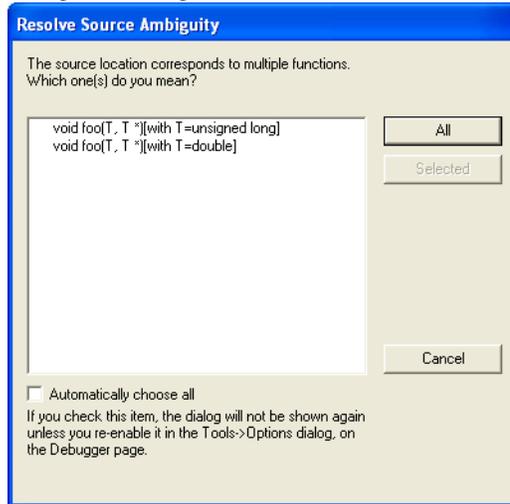
For example, `{C:\src\prog.c}.22.3` sets a breakpoint on the third character position on row 22 in the source file `prog.c`. Note that in quoted form, for example in a C-SPY macro, you must instead write

```
{C:\\src\\prog.c}.22.3.
```

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

Resolve Source Ambiguity dialog box

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



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

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

All

The breakpoint will be set on all listed locations.

Selected

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

Cancel

No location will be used.

Automatically choose all

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

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

Memory and registers

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

Introduction to monitoring memory and registers

These topics are covered:

- Briefly about monitoring memory and registers
- C-SPY memory zones
- Memory configuration for the C-SPY simulator
- Memory configuration for C-SPY hardware debugger drivers

BRIEFLY ABOUT MONITORING MEMORY AND REGISTERS

C-SPY provides many windows for monitoring memory and registers, most 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. *Data coverage* along with execution of your application is highlighted with different colors. 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, 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 **Registers** window

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

- The **SFR Setup** window

Displays the currently defined SFRs that C-SPY has information about, both factory-defined (retrieved from the device description file) and custom-defined SFRs. If required, you can use the **Edit SFR** dialog box to customize the SFR definitions.

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 **Registers** window containing any such registers is closed when debugging a running application.

C-SPY MEMORY ZONES

In C-SPY, the term *zone* is used for a named memory area. A memory address, or *location*, is a combination of a zone and a numerical offset into that zone. By default,

the RISC-V architecture has one memory-mapped zone, `Memory`, which covers the whole RISC-V memory range.



Default zone `Memory`

In addition to this zone, the control and status registers (CSRs) are accessed through the non memory-mapped `CSRMemory` zone.

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

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. When using the zone `Memory`, the debugger automatically selects the most suitable access width.

Note: For the C-SPY I-jet driver, you can specify the automatic selection of access width in the **Edit Memory Range** dialog box, see *Edit Memory Range dialog box for C-SPY hardware debugger drivers*, page 169.

MEMORY CONFIGURATION FOR THE C-SPY SIMULATOR

To simulate the target system properly, the C-SPY simulator needs information about the memory configuration. By default, C-SPY uses a configuration based on information retrieved from the device description file.

The C-SPY simulator provides various mechanisms to improve the configuration further:

- If the default memory configuration does not specify the required memory address ranges, you can specify the memory address ranges shall be based on:
 - The zones predefined in the device description file
 - The section information available in the debug file

- Or, you can define your own memory address ranges, which you typically might want to do if the files do not specify memory ranges for the *specific* device that you are using, but instead for a *family* of devices (perhaps with various amounts of on-chip RAM).
- For each memory address range, you can specify an *access type*. If a memory access occurs that does not agree with the specified access type, C-SPY will regard this as an illegal access and warn about it. In addition, an access to memory that is not defined is regarded as an illegal access. The purpose of memory access checking is to help you to identify memory access violations.

For more information, see *Memory Configuration dialog box for the C-SPY simulator*, page 162.

MEMORY CONFIGURATION FOR C-SPY HARDWARE DEBUGGER DRIVERS

To handle memory as efficiently as possible during debugging, C-SPY needs information about the memory configuration. By default, C-SPY uses a configuration based on information retrieved from the device description file.

You should make sure the memory address ranges match the memory available on your device. Providing C-SPY with information about the memory layout of the target system is helpful in terms of both performance and functionality:

- Reading (and writing) memory (if your debug probe is connected through a USB port) can be fast, but is usually the limiting factor when C-SPY needs to update many debugger windows. C-SPY can cache memory contents to speed up performance, provided it has correct information about the target memory.
- You can inform C-SPY that the content of certain memory address ranges will not be changed during a debug session. C-SPY can keep a copy of that memory readable even when the target system does not normally allow reading (such as when it is executing).

Note that if you specify the cache type **ROM/Flash**, C-SPY treats such memory as constant during the whole debug session (which improves efficiency, when updating some C-SPY windows). If your application modifies flash memory during runtime, do not use the **ROM/Flash** cache type.

- You can prevent C-SPY from accessing memory outside specified memory address ranges, which can be important for certain hardware.

The **Memory Configuration** dialog box is automatically displayed the first time you start the C-SPY driver for a given project, unless the device description file contains a memory description which is explicitly tagged as correct and complete. Subsequent starts will not display the dialog box unless you have made project changes that might

cause the memory configuration to change, for example if you have selected another device description file.

For more information, see *Memory Configuration dialog box for C-SPY hardware debugger drivers*, page 166.

Monitoring memory and registers

These tasks are covered:

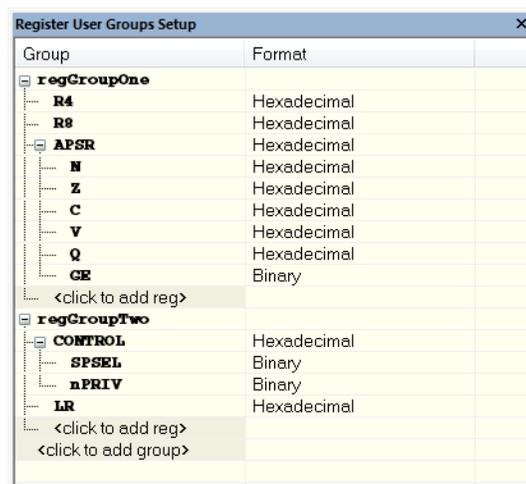
- Defining application-specific register groups
- Monitoring stack usage

DEFINING APPLICATION-SPECIFIC REGISTER GROUPS

Defining application-specific register groups minimizes the amount of registers displayed in the **Registers** windows and makes the debugging easier.

To define application-specific register groups:

- 1 Choose **View>Registers>Register User Groups Setup** during a debug session.



Right-clicking in the window displays a context menu with commands. For information about these commands, see *Register User Groups Setup window*, page 155.

- 2 Click on **<click to add group>** and specify the name of your group, for example **My Timer Group** and press Enter.

- 3 Underneath the group name, click on <click to add reg> and type the name of a register, and press Enter. You can also drag a register name from another window in the IDE. Repeat this for all registers that you want to add to your group.
- 4 As an optional step, right-click any registers for which you want to change the integer base, and choose **Format** from the context menu to select a suitable base.
- 5 When you are done, your new group is now available in the **Registers** windows.

If you want to define more application-specific groups, repeat this procedure for each group you want to define.

Note: If a certain SFR that you need cannot be added to a group, you can register your own SFRs. For more information, see *SFR Setup window*, page 157.

MONITORING STACK USAGE

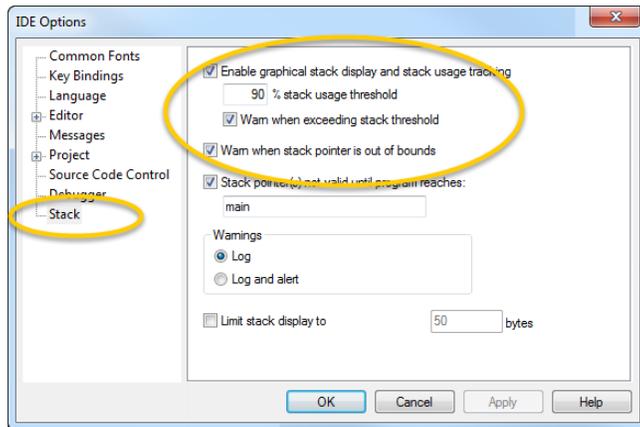
These are the two main use cases for the **Stack** window:

- Monitoring stack memory usage
- Monitoring the stack memory content.

In both cases, C-SPY retrieves information about the defined stack size and its allocation from the definition in the linker configuration file of the section holding the stack. If you, for some reason, have modified the stack initialization 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, see the *IAR C/C++ Development Guide for RISC-V*.

To monitor stack memory usage:

- 1 Before you start C-SPY, choose **Tools>Options**. On the **Stack** page:
 - Select **Enable graphical stack display and stack usage tracking**. This option also enables the option **Warn when exceeding stack threshold**. Specify a suitable threshold value.
 - Note also the option **Warn when stack pointer is out of bounds**. Any such warnings are displayed in the **Debug Log** window.



2 Start C-SPY.

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.

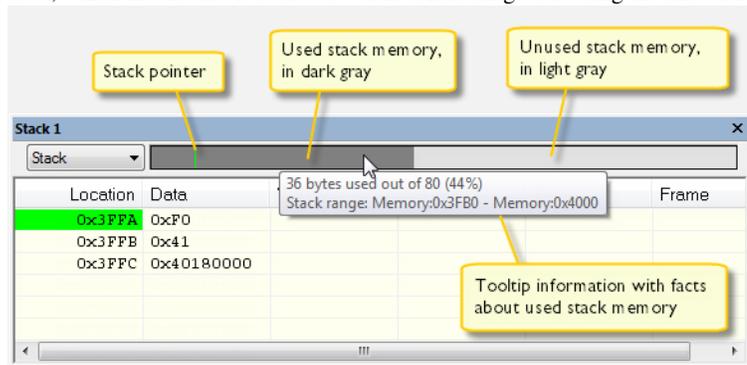
3 Choose **View>Stack>Stack 1** to open the **Stack** window.

Note that 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.

4 Start executing your application.

Whenever execution stops, the stack memory is searched from the end of the stack until a byte whose value is not `0xCD` is found, which is assumed to be how far the stack has been used. The light gray area of the stack bar represents the *unused* stack memory area, whereas the dark gray area of the bar represents the *used* stack memory.

For this example, you can see that only 44% of the reserved memory address range was used, which means that it could be worth considering decreasing the size of memory:



Note: 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 end of the stack range. Likewise, your application might modify memory within the stack area by mistake.

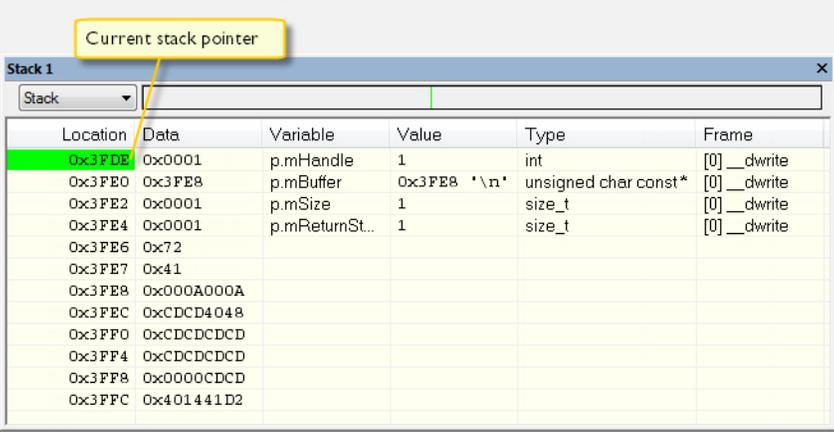
To monitor the stack memory content:

- 1 Before you start monitoring stack memory, you might want to disable the option **Enable graphical stack display and stack usage tracking** to improve performance during debugging.
- 2 Start C-SPY.
- 3 Choose **View>Stack>Stack 1** to open the **Stack** window.

Note that you can access various context menus in the display area from where you can change display format, etc.

- 4 Start executing your application.

Whenever execution stops, you can monitor the stack memory, for example to see function parameters that are passed on the stack:



Location	Data	Variable	Value	Type	Frame
0x3FDE	0x0001	p.mHandle	1	int	[0] __dwrite
0x3FE0	0x3FE8	p.mBuffer	0x3FE8 '\n'	unsigned char const*	[0] __dwrite
0x3FE2	0x0001	p.mSize	1	size_t	[0] __dwrite
0x3FE4	0x0001	p.mReturnSt...	1	size_t	[0] __dwrite
0x3FE6	0x72				
0x3FE7	0x41				
0x3FE8	0x000A000A				
0x3FEC	0xCDCD4048				
0x3FF0	0xCDCDCDCD				
0x3FF4	0xCDCDCDCD				
0x3FF8	0x0000CDCD				
0x3FFC	0x401441D2				

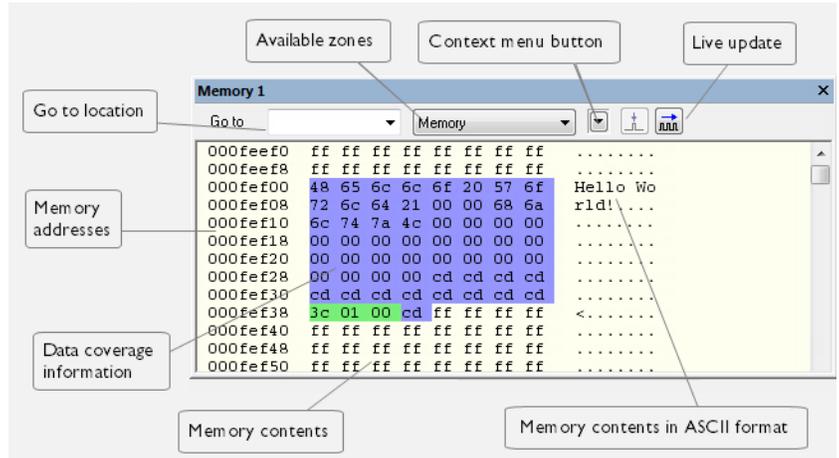
Reference information on memory and registers

Reference information about:

- *Memory window*, page 138
- *Memory Save dialog box*, page 142
- *Memory Restore dialog box*, page 143
- *Fill dialog box*, page 144
- *Symbolic Memory window*, page 145
- *Stack window*, page 148
- *Registers window*, page 152
- *Register User Groups Setup window*, page 155
- *SFR Setup window*, page 157
- *Edit SFR dialog box*, page 160
- *Memory Configuration dialog box for the C-SPY simulator*, page 162
- *Edit Memory Range dialog box for the C-SPY simulator*, page 164
- *Memory Configuration dialog box for C-SPY hardware debugger drivers*, page 166
- *Edit Memory Range dialog box for C-SPY hardware debugger drivers*, page 169

Memory window

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



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



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

See also *Editing in C-SPY windows*, page 46.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Context menu button

Displays the context menu.

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.

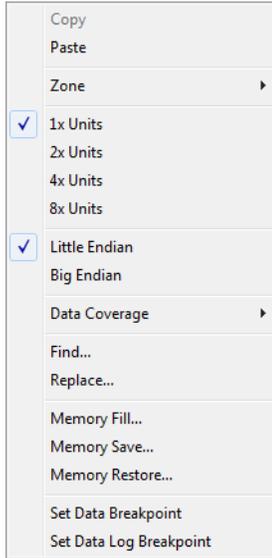
Data coverage is displayed with these colors:

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:



These commands are available:

Copy

Paste

Standard editing commands.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

1x Units

Displays the memory contents as single bytes.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

8x Units

Displays the memory contents as 8-byte groups.

Little Endian

Displays the contents in little-endian byte order.

Big Endian

Displays the contents in big-endian byte order.

Data Coverage

Choose between:

Enable toggles data coverage on or off.

Show toggles between showing or hiding data coverage.

Clear clears all data coverage information.

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

Find

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

Replace

Displays a dialog box where you can search for a specified string and replace each occurrence with another string—read about the **Replace** dialog box in the *IDE Project Management and Building Guide for RISC-V*.

Memory Fill

Displays a dialog box, where you can fill a specified area with a value, see *Fill dialog box*, page 144.

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 142.

Memory Restore

Displays a dialog box, where you can load the contents of a file in Intel-hex or Motorola s-record format to a specified memory zone, see *Memory Restore dialog box*, page 143.

Set Data Breakpoint

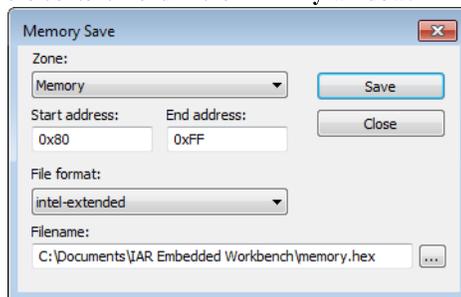
Sets breakpoints directly in the **Memory** window. The breakpoint is not highlighted—you can see, edit, and remove it in the breakpoint dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the Memory window*, page 112.

Set Data Log Breakpoint

Sets a breakpoint on the start address of a memory selection directly in the **Memory** window. The breakpoint is not highlighted—you can see, edit, and remove it in the breakpoint dialog box. The breakpoints you set in this window will be triggered by both read and write accesses—to change this, use the **Breakpoints** window. For more information, see *Data Log breakpoints*, page 107 and *Getting started using data logging*, page 201.

Memory Save dialog box

The **Memory Save** dialog box is available by choosing **Debug>Memory>Save** or from the context menu in the **Memory** window.



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

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

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

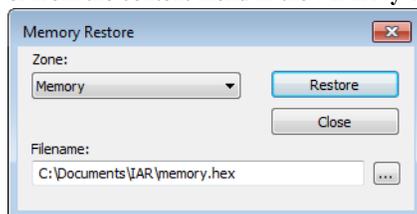
Specify the destination file to be used. A browse button is available.

Save

Saves the selected range of the memory zone to the specified file.

Memory Restore dialog box

The **Memory Restore** dialog box is available by choosing **Debug>Memory>Restore** or from the context menu in the **Memory** window.



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

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Filename

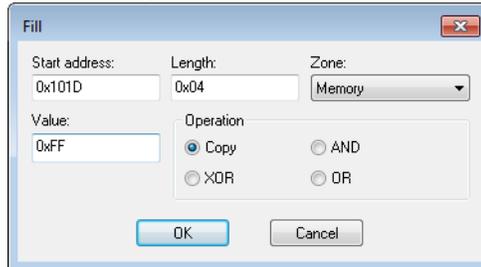
Specify the file to be read. A browse button is available.

Restore

Loads the contents of the specified file to the selected memory zone.

Fill dialog box

The **Fill** dialog box is available from the context menu in the **Memory** window.



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

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Start address

Type the start address—in binary, octal, decimal, or hexadecimal notation.

Length

Type the length—in binary, octal, decimal, or hexadecimal notation.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Value

Type the 8-bit value to be used for filling each memory location.

Operation

These are the available memory fill operations:

Copy

Value will be copied to the specified memory area.

AND

An **AND** operation will be performed between Value and the existing contents of memory before writing the result to memory.

XOR

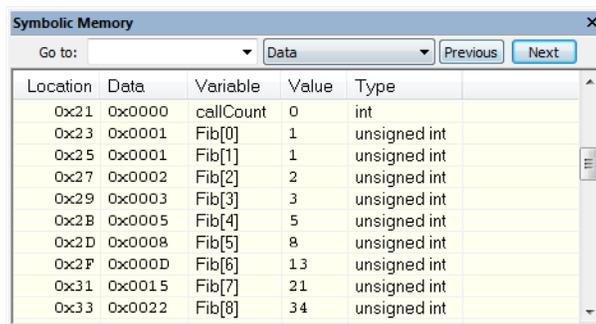
An **XOR** operation will be performed between Value and the existing contents of memory before writing the result to memory.

OR

An OR operation will be performed between Value and the existing contents of memory before writing the result to memory.

Symbolic Memory window

The **Symbolic Memory** window is available from the **View** menu during a debug session.



Location	Data	Variable	Value	Type
0x21	0x0000	callCount	0	int
0x23	0x0001	Fib[0]	1	unsigned int
0x25	0x0001	Fib[1]	1	unsigned int
0x27	0x0002	Fib[2]	2	unsigned int
0x29	0x0003	Fib[3]	3	unsigned int
0x2B	0x0005	Fib[4]	5	unsigned int
0x2D	0x0008	Fib[5]	8	unsigned int
0x2F	0x000D	Fib[6]	13	unsigned int
0x31	0x0015	Fib[7]	21	unsigned int
0x33	0x0022	Fib[8]	34	unsigned int

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.

See also *Editing in C-SPY windows*, page 46.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

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.

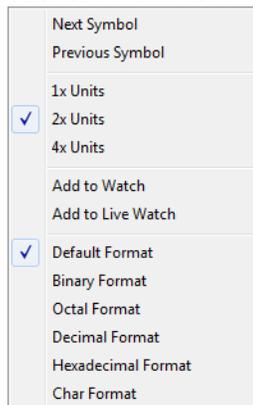
There are several different ways to navigate within the memory space:

- Text that is dropped in the window is interpreted as symbols
- The scroll bar at the right-side of the window
- The **Next** and **Previous** toolbar buttons
- The **Go to** toolbar list box can be used for locating specific locations or symbols.

Note: Rows are marked in red when the corresponding value has changed.

Context menu

This context menu is available:



These commands are available:

Next Symbol

Highlights the next symbol in the display area.

Previous Symbol

Highlights the previous symbol in the display area.

1x Units

Displays the memory contents as single bytes. This applies only to rows that do not contain a variable.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

Add to Watch

Adds the selected symbol to the **Watch** window.

Add to Live Watch

Adds the selected symbol to the **Live Watch** window.

Default format

Displays the memory contents in the default format.

Binary format

Displays the memory contents in binary format.

Octal format

Displays the memory contents in octal format.

Decimal format

Displays the memory contents in decimal format.

Hexadecimal format

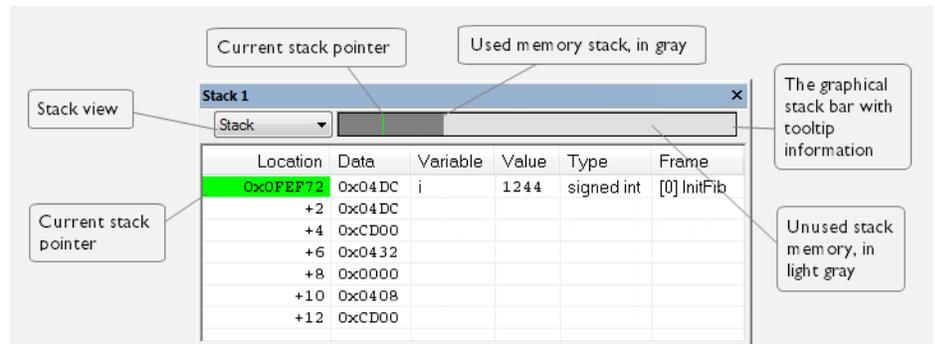
Displays the memory contents in hexadecimal format.

Char format

Displays the memory contents in char format.

Stack window

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



This window is a memory window that displays the contents of the stack. The graphical stack bar shows stack usage.

Note: By default, this window uses one physical breakpoint. For more information, see *Breakpoint consumers*, page 109.

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

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Toolbar

The toolbar contains:

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 address range reserved for the stack. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

To enable the stack bar, choose **Tools>Options>Stack>Enable graphical stack display and stack usage tracking**. This means that the functionality needed to detect and warn about stack overflows is 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.

Type

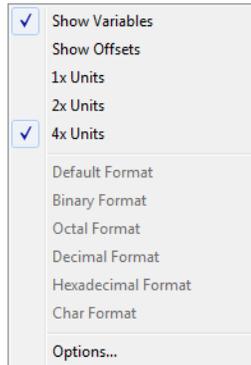
Displays the data type of the variable.

Frame

Displays the name of the function that the call frame corresponds to.

Context menu

This context menu is available:



These commands are available:

Show Variables

Displays separate columns named **Variables**, **Value**, and **Frame** in the **Stack** window. Variables located at memory addresses listed in the **Stack** window are displayed in these columns.

Show Offsets

Displays locations in the **Location** column as offsets from the stack pointer. When deselected, locations are displayed as absolute addresses.

1x Units

Displays the memory contents as single bytes.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

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

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

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

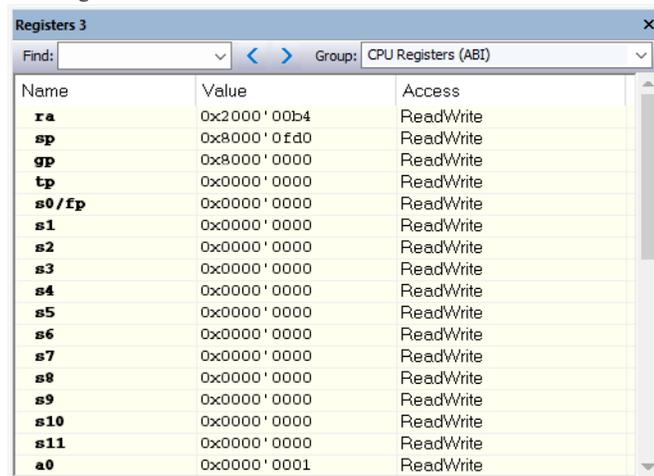
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration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

Options

Opens the **IDE Options** dialog box where you can set options specific to the **Stack** window, see the *IDE Project Management and Building Guide for RISC-V*.

Registers window

The **Registers** windows are available from the **View** menu.



Name	Value	Access
ra	0x2000'00b4	ReadWrite
sp	0x8000'0fd0	ReadWrite
gp	0x8000'0000	ReadWrite
tp	0x0000'0000	ReadWrite
s0/Ep	0x0000'0000	ReadWrite
s1	0x0000'0000	ReadWrite
s2	0x0000'0000	ReadWrite
s3	0x0000'0000	ReadWrite
s4	0x0000'0000	ReadWrite
s5	0x0000'0000	ReadWrite
s6	0x0000'0000	ReadWrite
s7	0x0000'0000	ReadWrite
s8	0x0000'0000	ReadWrite
s9	0x0000'0000	ReadWrite
s10	0x0000'0000	ReadWrite
s11	0x0000'0000	ReadWrite
a0	0x0000'0001	ReadWrite

These windows give an up-to-date display of the contents of the processor registers and special function registers, and allow you to edit the contents of some of the registers. Optionally, you can choose to load either predefined register groups or your own user-defined groups.

You can open up to four instances of this window, which is convenient for keeping track of different register groups.

See also *Editing in C-SPY windows*, page 46.

To enable predefined register groups:

- 1 Select a device description file that suits your device, see *Selecting a device description file*, page 43. These files contain predefined register groups.
- 2 Display the registers of a register group by selecting it from the **Group** drop-down menu on the toolbar, or by right-clicking in the window and choosing **View Group** from the context menu.

For information about creating your own user-defined register groups, see *Defining application-specific register groups*, page 133.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Toolbar

The toolbar contains:

Find

Specify the name, or part of a name, of a register (or group) that you want to find. Press the Enter key and the first matching register, or group with a matching register, is displayed. User-defined register groups are not searched. The search box preserves a history of previous searches. To repeat a search, select it from the search history and press Enter.

Group

Selects which predefined register group to display. Additional register groups are predefined in the device description files that make SFR registers available in the **Registers** windows. The device description file contains a section that defines the special function registers and their groups. If some of your SFRs are missing, you can register your own SFRs in a Custom group, see *SFR Setup window*, page 157.

Display area

Displays registers and their values. Some registers are expandable, which means that the register contains interesting bits or subgroups of bits.

If you drag a numerical value, a valid expression, or a register name from another part of the IDE to an editable value cell in a **Registers** window, the value will be changed to that of what you dragged. If you drop a register name somewhere else in the window, the window contents will change to display the first register group where this register is found.

Name

The name of the register.

Value

The current value of the register. Every time C-SPY stops, a value that has changed since the last stop is highlighted. Some of the registers are editable. To edit the contents of an editable register, click on the register and modify its value. Press Esc to cancel the change.

To change the display format of the value, right-click on the register and choose **Format** from the context menu.

Access

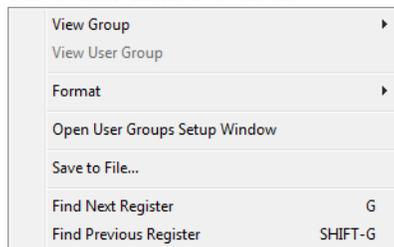
The access type of the register. Some of the registers are read-only, while others are write-only.

For the C-SPY Simulator, these additional support registers are available in the CPU Registers group:

CYCLECOUNTER	Cleared when an application is started or reset, and is incremented with the number of used cycles during execution.
CCSTEP	Shows the number of used cycles during the last performed C/C++ source or assembler step.
CCTIMER1 and CCTIMER2	Two <i>trip counts</i> that can be cleared manually at any given time. They are incremented with the number of used cycles during execution.

Context menu

This context menu is available:



These commands are available:

View Group

Selects which predefined register group to display.

View User Group

Selects which user-defined register group to display. For information about creating your own user-defined register groups, see *Defining application-specific register groups*, page 133.

Format

Changes the display format for the contents of the register you clicked on. The display format setting affects different types of registers in different ways. Your selection of display format is saved between debug sessions.

Open User Groups Setup Window

Opens a window where you can create your own user-defined register groups, see *Register User Groups Setup window*, page 155.

Save to File

Opens a standard **Save** dialog box to save the contents of the window to a tab-separated text file.

Find Next Register

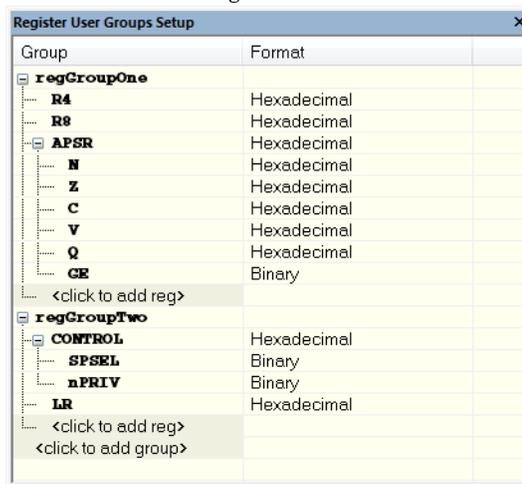
Finds the predefined register or register group that comes immediately after what your search found. After the last register was found, this search wraps around and finds the first register again.

Find Previous Register

Finds the matching predefined register or register group that comes immediately before what your search found. After the first register was found, this search wraps around and finds the last register again.

Register User Groups Setup window

The **Register User Groups Setup** window is available from the **View** menu or from the context menu in the **Registers** windows.



Use this window to define your own application-specific register groups. These register groups can then be viewed in the **Registers** windows.

Defining application-specific register groups means that the **Registers** windows can display just those registers that you need to watch for your current debugging task. This makes debugging much easier.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

This area contains these columns:

Group

The names of register groups and the registers they contain. Clicking on <click to add group> or <click to add reg> and typing the name of a register group or register, adds new groups and registers, respectively. You can also drag a register name from another window in the IDE. Click a name to change it.

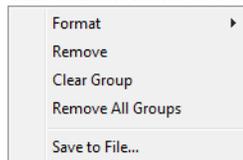
A dimmed register name indicates that it is not supported by the selected device.

Format

Shows the display format for the register's value. To change the display format of the value, right-click on the register and choose **Format** from the context menu. The selected format is used in all **Registers** windows.

Context menu

This context menu is available:



These commands are available:

Format

Changes the display format for the contents of the register you clicked on. The display format setting affects different types of registers in different ways. Your selection of display format is saved between debug sessions.

Remove

Removes the register or group you clicked on.

Clear Group

Removes all registers from the group you clicked on.

Remove All Groups

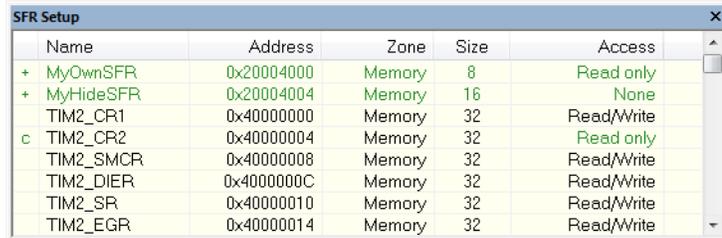
Deletes all user-defined register groups from your project.

Save to File

Opens a standard **Save** dialog box to save the contents of the window to a tab-separated text file.

SFR Setup window

The **SFR Setup** window is available from the **Project** menu.



Name	Address	Zone	Size	Access
+ MyOwnSFR	0x20004000	Memory	8	Read only
+ MyHideSFR	0x20004004	Memory	16	None
TIM2_CR1	0x40000000	Memory	32	Read/Write
c TIM2_CR2	0x40000004	Memory	32	Read only
TIM2_SMCR	0x40000008	Memory	32	Read/Write
TIM2_DIER	0x4000000C	Memory	32	Read/Write
TIM2_SR	0x40000010	Memory	32	Read/Write
TIM2_EGR	0x40000014	Memory	32	Read/Write

This window displays the currently defined SFRs that C-SPY has information about. You can choose to display only factory-defined or custom-defined SFRs, or both. If required, you can use the **Edit SFR** dialog box to customize the SFR definitions, see *Edit SFR dialog box*, page 160. For factory-defined SFRs (that is, retrieved from the `adf` file in use), you can only customize the access type.

To quickly find an SFR, drag a text or hexadecimal number string and drop in this window. If what you drop starts with a 0 (zero), the **Address** column is searched, otherwise the **Name** column is searched.

Any custom-defined SFRs are added to a dedicated register group called Custom, which you can choose to display in the **Registers** window. Your custom-defined SFRs are saved in `projectCustomSFR.sfr`. This file is automatically loaded in the IDE when you start C-SPY with a project whose name matches the prefix of the filename of the `sfr` file.

You can only add or modify SFRs when the C-SPY debugger is not running.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

This area contains these columns:

Status

A character that signals the status of the SFR, which can be one of:
blank, a factory-defined SFR.

C, a factory-defined SFR that has been modified.

+, a custom-defined SFR.

?, an SFR that is ignored for some reason. An SFR can be ignored when a factory-defined SFR has been modified, but the SFR is no longer available, or it is located somewhere else or with a different size. Typically, this might happen if you change to another device.

Name

A unique name of the SFR.

Address

The memory address of the SFR.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Size

The size of the register, which can be any of **8**, **16**, **32**, or **64**.

Access

The access type of the register, which can be one of **Read/Write**, **Read only**, **Write only**, or **None**.

You can click a name or an address to change the value. The hexadecimal 0x prefix for the address can be omitted, the value you enter will still be interpreted as hexadecimal. For example, if you enter 4567, you will get 0x4567.

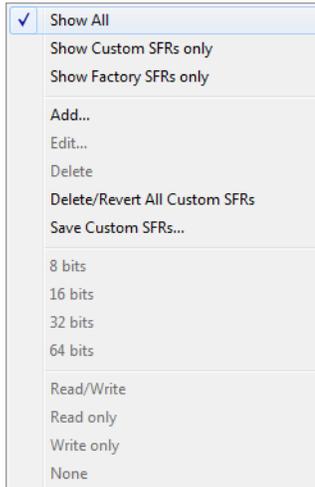
You can click a column header to sort the SFRs according to the column property.

Color coding used in the display area:

- Green, which indicates that the corresponding value has changed
- Red, which indicates an ignored SFR.

Context menu

This context menu is available:



These commands are available:

Show All

Shows all SFR.

Show Custom SFRs only

Shows all custom-defined SFRs.

Show Factory SFRs only

Shows all factory-defined SFRs retrieved from the ddf file.

Add

Displays the **Edit SFR** dialog box where you can add a new SFR, see *Edit SFR dialog box*, page 160.

Edit

Displays the **Edit SFR** dialog box where you can edit an SFR, see *Edit SFR dialog box*, page 160.

Delete

Deletes an SFR. This command only works on custom-defined SFRs.

Delete/Revert All Custom SFRs

Deletes all custom-defined SFRs and reverts all modified factory-defined SFRs to their factory settings.

Save Custom SFRs

Opens a standard **Save** dialog box to save all custom-defined SFRs.

8|16|32|64 bits

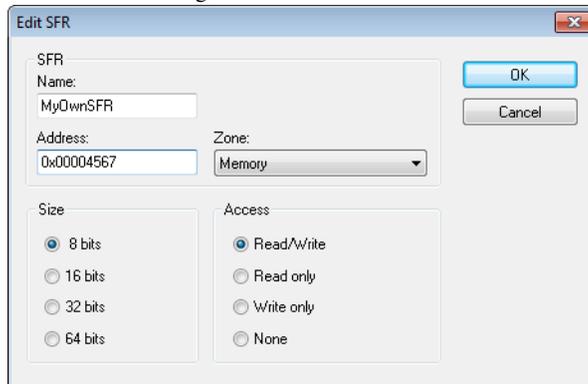
Selects display format for the selected SFR, which can be **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.

Read/Write|Read only|Write only|None

Selects the access type of the selected SFR, which can be **Read/Write**, **Read only**, **Write only**, or **None**. Note that for factory-defined SFRs, the default access type is indicated.

Edit SFR dialog box

The **Edit SFR** dialog box is available from the context menu in the **SFR Setup** window.



Definitions of the SFRs are retrieved from the device description file in use. Use this dialog box to either modify these factory-defined definitions or define new SFRs. See also *SFR Setup window*, page 157.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Name

Specify the name of the SFR that you want to add or edit.

Address

Specify the address of the SFR that you want to add or edit. The hexadecimal 0x prefix for the address can be omitted, the value you enter will still be interpreted as hexadecimal. For example, if you enter 4567, you will get 0x4567.

Zone

Selects the memory zone for the SFR you want to add or edit. The list of zones is retrieved from the `ddf` file that is currently used.

Size

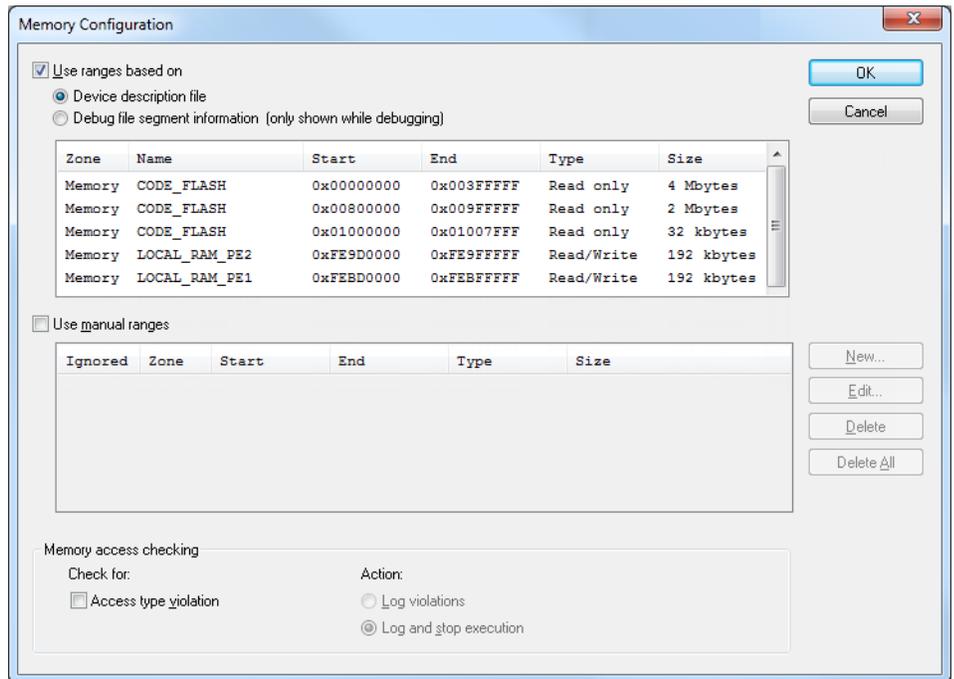
Selects the size of the SFR. Choose between **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.

Access

Selects the access type of the SFR. Choose between **Read/Write**, **Read only**, **Write only**, or **None**. Note that for factory-defined SFRs, the default access type is indicated.

Memory Configuration dialog box for the C-SPY simulator

The Memory Configuration dialog box is available from the C-SPY driver menu.



Use this dialog box to specify which set of memory address ranges to be used by C-SPY during debugging.

See also *Memory configuration for the C-SPY simulator*, page 131.

Requirements

The C-SPY simulator.

Use ranges based on

Specify if the memory configuration should be retrieved from a predefined configuration. Choose between:

Device description file

Retrieves the memory configuration from the device description file that you have specified. See *Selecting a device description file*, page 43.

This option is used by default.

Debug file segment information

Retrieves the memory configuration from the debug file, which has retrieved it from the linker configuration file. This information is only available during a debug session. The advantage of using this option is that the simulator can catch memory accesses outside the linked application.

Memory information is displayed in these columns:

Zone

The memory zone, see *C-SPY memory zones*, page 130.

Name

The name of the memory address range.

Start

The start address for the memory address range, in hexadecimal notation.

End

The end address for the memory address range, in hexadecimal notation.

Type

The access type of the memory address range.

Size

The size of the memory address range.

Use manual ranges

Specify your own ranges manually via the **Edit Memory Range** dialog box. To open this dialog box, click **New** to specify a new memory address range, or select an existing memory address range and click **Edit** to modify it. For more information, see *Edit Memory Range dialog box for the C-SPY simulator*, page 164.

The ranges you define manually are saved between debug sessions.

An **X** in the column **Ignored** means that C-SPY has detected that the specified manual range is illegal, for example because it overlaps another range. C-SPY will not use such an area.

Memory access checking

Check for determines what to check for:

- **Access type violation.**

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 for the manual ranges:

New

Opens the **Edit Memory Range** dialog box, where you can specify a new memory address range and associate an access type with it, see *Edit Memory Range dialog box for the C-SPY simulator*, page 164.

Edit

Opens the **Edit Memory Range** dialog box, where you can edit the selected memory address range. See *Edit Memory Range dialog box for the C-SPY simulator*, page 164.

Delete

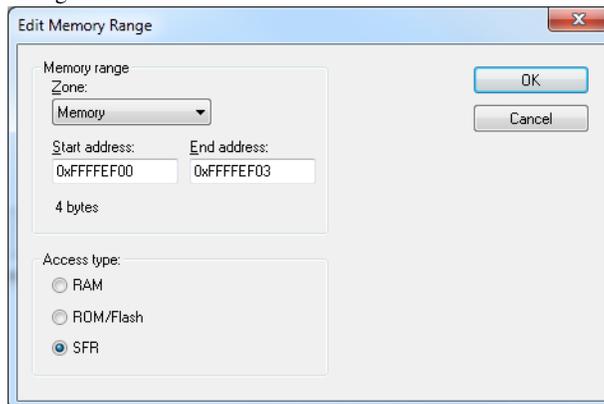
Deletes the selected memory address range definition.

Delete All

Deletes all defined memory address range definitions.

Edit Memory Range dialog box for the C-SPY simulator

The **Edit Memory Range** dialog box is available from the **Memory Configuration** dialog box.



Use this dialog box to specify your own memory address ranges, and their access types.

See also *Memory Configuration dialog box for the C-SPY simulator*, page 162

Requirements

The C-SPY simulator.

Memory range

Defines the memory address range specific to your device:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Start address

Specify the start address for the memory address range, in hexadecimal notation.

End address

Specify the end address for the memory address range, in hexadecimal notation.

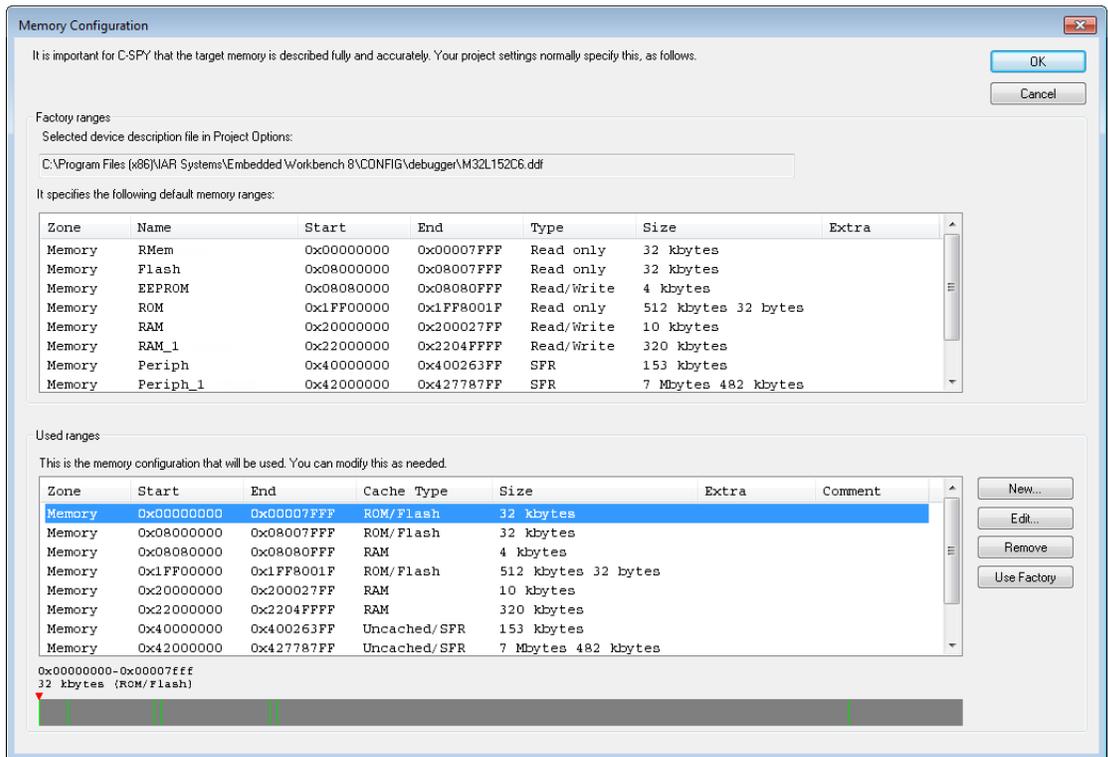
Access type

Selects an access type for the memory address range. Choose between:

- **RAM**, for read/write memory
- **ROM/Flash**, for read-only memory
- **SFR**, for SFR read/write memory.

Memory Configuration dialog box for C-SPY hardware debugger drivers

The Memory Configuration dialog box is available from the C-SPY driver menu.



C-SPY uses a default memory configuration based on information retrieved from the device description file in use. If memory configuration is missing in the device description file, C-SPY tries to provide a usable factory default. See *Selecting a device description file*, page 43.

Use this dialog box to verify, and if needed, modify the memory address ranges so that they match the memory available on your device.

You can only change the memory configuration when C-SPY is not running.

See also *Memory configuration for C-SPY hardware debugger drivers*, page 132.

Requirements

Any C-SPY hardware debugger driver.

Factory ranges

Identifies which device description file that is currently selected and lists the default memory address ranges retrieved from the file in these columns:

Zone

The memory zone, see *C-SPY memory zones*, page 130.

Name

The name of the memory address range.

Start

The start address for the memory address range, in hexadecimal notation.

End

The end address for the memory address range, in hexadecimal notation.

Type

The access type of the memory address range.

Size

The size of the memory address range.

Used ranges

These columns list the memory address ranges that will be used by C-SPY. The columns are normally identical to the factory ranges, unless you have added, removed, or modified ranges.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Start

The start address for the memory address range, in hexadecimal notation.

End

The end address for the memory address range, in hexadecimal notation.

Cache Type

The cache type of the memory address range.

Size

The size of the memory address range.

Comment

Memory area information.

Use the buttons to override the default memory address ranges that are retrieved from the device description file.

Graphical bar

A graphical bar that visualizes the entire theoretical memory address range for the device. Defined ranges are highlighted in green.

Buttons

These buttons are available for manual ranges:

New

Opens the **Edit Memory Range** dialog box, where you can specify a new memory address range and associate a cache type with it, see *Edit Memory Range dialog box for C-SPY hardware debugger drivers*, page 169.

Edit

Opens the **Edit Memory Range** dialog box, where you can edit the selected memory address area. See *Edit Memory Range dialog box for C-SPY hardware debugger drivers*, page 169.

Remove

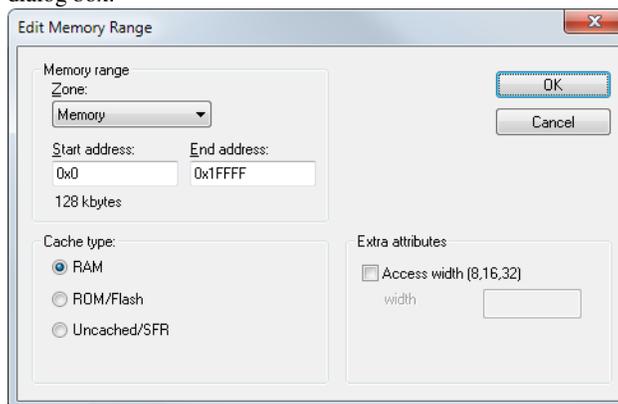
Removes the selected memory address range definition.

Use Factory

Restores the list of used ranges to the factory ranges.

Edit Memory Range dialog box for C-SPY hardware debugger drivers

The Edit Memory Range dialog box is available from the Memory Configuration dialog box.



Use this dialog box to specify the memory address ranges, and assign a cache type to each range.

See also *Memory configuration for C-SPY hardware debugger drivers*, page 132.

Requirements

Any C-SPY hardware debugger driver.

Memory range

Defines the memory address range specific to your device:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 130.

Start address

Specify the start address for the memory address range, in hexadecimal notation.

End address

Specify the end address for the memory address range, in hexadecimal notation.

Cache type

Selects a cache type to the memory address range. Choose between:

RAM

When the target CPU is not executing, all read accesses from memory are loaded into the cache. For example, if two **Memory** windows show the same part of memory, the actual memory is only read once from the hardware to update both windows. If you modify memory from a C-SPY window, your data is written to cache only. Before any target execution, even stepping a single machine instruction, the RAM cache is flushed so that all modified bytes are written to the memory on your hardware.

ROM/Flash

This memory is assumed not to change during a debug session. Any code within such a range that is downloaded when you start a debug session (or technically, any such code that is part of the application being debugged) is stored in the cache and remains there. Other parts of such ranges are loaded into the cache from memory on demand, but are then kept during the debug session. Note that C-SPY will not allow you to modify such memory from C-SPY windows.

Even though flash memory is normally used as a fixed read-only memory, there are applications that modify parts of flash memory at runtime. For example, some part of flash memory might be used for a file system or simply to store non-volatile information. To reflect this in C-SPY, you should choose the **RAM** cache type for those instead. Then C-SPY will assume that those parts can change at any time during execution.

SFR/Uncached

A range of this type is completely uncached. All read or write commands from a C-SPY window will access the hardware immediately. Typically, this type is useful for special function registers, which can have all sorts of unusual behavior, such as having different values at every read access. This can in turn have side-effects on other registers when they are written, not containing the same value as was previously written, etc.

If you do not have the appropriate information about your device, you can specify an entire memory as **SFR/Uncached**. This is not incorrect, but might make C-SPY slower when updating windows. In fact, this caching type is sometimes used by the default when there is no memory address range information available.

If required, you can disable caching—choose *C-SPY driver*>**Disable Debugger Cache**.

Extra attributes

Provides extra attributes.

Access width [8,16,32,64]

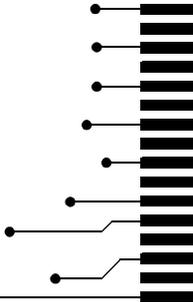
Forces C-SPY to use 8, 16, 32, or 64-bit width when accessing memory in this range. Specify 8, 16, 32, or 64 in the text box.

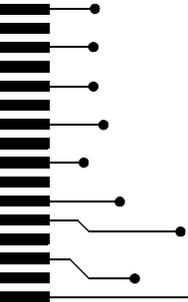
This option might not be available in the C-SPY driver you are using.

Part 2. Analyzing your application

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

- Trace
- The application timeline
- Profiling
- Code coverage





Trace

- Introduction to using trace
- Collecting and using trace data
- Reference information on trace

Introduction to using trace

These topics are covered:

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

See also:

- *Getting started using data logging*, page 201
- *Getting started using interrupt logging*, page 250
- *Profiling*, page 215

REASONS FOR USING TRACE

By using trace, you can inspect the program flow up to a specific state, for instance an application crash, and use the trace data to locate the origin of the problem. Trace data can be useful for locating programming errors that have irregular symptoms and occur sporadically.

BRIEFLY ABOUT TRACE

To use trace in C-SPY requires that your target system can generate trace data. Once generated, C-SPY can collect it and you can visualize and analyze the data in various windows and dialog boxes.

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

- SiFive devices with support for Nexus-based trace
- The C-SPY simulator

External trace

External real-time trace is a continuously collected sequence of every executed instruction for a selected portion of the execution. The trace buffer is located in the debug probe and is continuously consumed by the C-SPY driver. However, if the data rate is too high, the data transfer will stall and the trace buffer could eventually fill up. The trace buffer collects trace data in real time, but the data is not displayed in the C-SPY windows until after the execution has stopped.

Note: Collecting this type of trace data requires an I-jet Trace debug probe.

RAM trace

RAM trace uses a designated on-chip trace buffer. The trace buffer collects trace data in real time, but the data is not displayed in the C-SPY windows until after the execution has stopped.

Note: The amount of trace data (including code coverage and function profiling) that can be collected is limited by the size of the on-chip trace RAM buffer.

Serial trace

Serial trace is a sequence of events of various kinds, generated by the on-chip debug hardware. The most important events are:

- *PC sampling*
The hardware can sample and transmit the value of the program counter at regular intervals.
- *Interrupt logs*
The hardware can generate and transmit data related to the execution of interrupts.
- *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.

Note: Collecting this type of trace data requires a combination of debug probe and target hardware that support it.

Trace features in C-SPY

In C-SPY, you can use the trace-related windows—**Trace**, **Function Trace**, **Timeline**, and **Find in Trace**.

REQUIREMENTS FOR USING TRACE

The C-SPY simulator supports trace-related functionality, and there are no specific requirements.

Requirements for using Nexus-based trace

To use Nexus-based trace, you need an I-jet or I-jet Trace debug probe and an appropriate adapter. SRAM-based and External trace is supported. External trace requires an I-jet Trace debug probe.

Collecting and using trace data

These tasks are covered:

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

GETTING STARTED WITH TRACE

To collect trace data, no specific build settings are required.

- 1 Start C-SPY and choose **I-jet>Trace Settings**. In the **Trace Settings** dialog box that is displayed, check if you need to change any of the default settings and set the **Mode** option to the type of trace you want to collect. If you made any changes, exit C-SPY and restart the debug session.

For more information about the **Trace Settings** dialog box, see *Trace Settings dialog box*, page 180.

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



- 2 Open the **Trace** window—available from the driver-specific menu—and click the **Activate** button to enable collecting trace data.
- 3 Start the execution. When the execution stops, for example because a breakpoint is triggered, trace data is displayed in the **Trace** window. For more information about the window, see *Trace window*, page 183.

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 Trigger** or **Trace Stop Trigger** breakpoint from the context menu.
- In the **Breakpoints** window, choose **New Breakpoint>Trace Start Trigger** or **Trace Stop Trigger** from the context menu.
- The C-SPY system macros `__setTraceStartBreak` and `__setTraceStopBreak` can also be used.

For more information about these breakpoints, see *Trace Start Trigger breakpoint dialog box*, page 191 and *Trace Stop Trigger breakpoint dialog box*, page 193, respectively.

SEARCHING IN TRACE DATA

When you have collected trace data, you can perform searches in the collected data to locate the parts of your code or data that you are interested in, for example, a specific interrupt or accesses of a specific variable.

You specify the search criteria in the **Find in Trace** dialog box and view the result in the **Find in Trace** window.

The **Find in Trace** window is very similar to the **Trace** window, showing the same columns and data, but *only* those rows that match the specified search criteria. Double-clicking an item in the **Find in Trace** window brings up the same item in the **Trace** window.

To search in your trace data:



- 1 On the **Trace** window toolbar, click the **Find** button.
- 2 In the **Find in Trace** dialog box, specify your search criteria.

Typically, you can choose to search for:

- A specific piece of text, for which you can apply further search criteria
- An address range
- A combination of these, like a specific piece of text within a specific address range.

For more information about the various options, see *Find in Trace dialog box*, page 194.

- 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 195.

BROWSING THROUGH TRACE DATA

To follow the execution history, simply look and scroll in the **Trace** window. Alternatively, you can enter *browse mode*.



To enter browse mode, double-click an item in the **Trace** window, or click the **Browse** toolbar button.

The selected item turns yellow and the source and **Disassembly** windows will highlight the corresponding location. You can now move around in the trace data using the up and down arrow keys, or by scrolling and clicking—the source and **Disassembly** windows will be updated to show the corresponding location. This is like stepping backward and forward through the execution history.

Double-click again to leave browse mode.

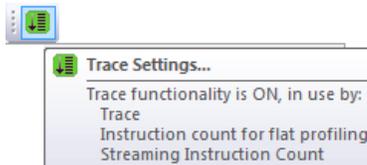
Reference information on trace

Reference information about:

- *Trace Settings button in the IDE toolbar*, page 179
- *Trace Settings dialog box*, page 180
- *Trace window*, page 183
- *Function Trace window*, page 190
- *Trace Start Trigger breakpoint dialog box*, page 191
- *Trace Stop Trigger breakpoint dialog box*, page 193
- *Find in Trace dialog box*, page 194
- *Find in Trace window*, page 195

Trace Settings button in the IDE toolbar

During a debug session using a C-SPY driver, a debug probe, and a device that all support trace, this button is added to the toolbar in the main Embedded Workbench IDE window:

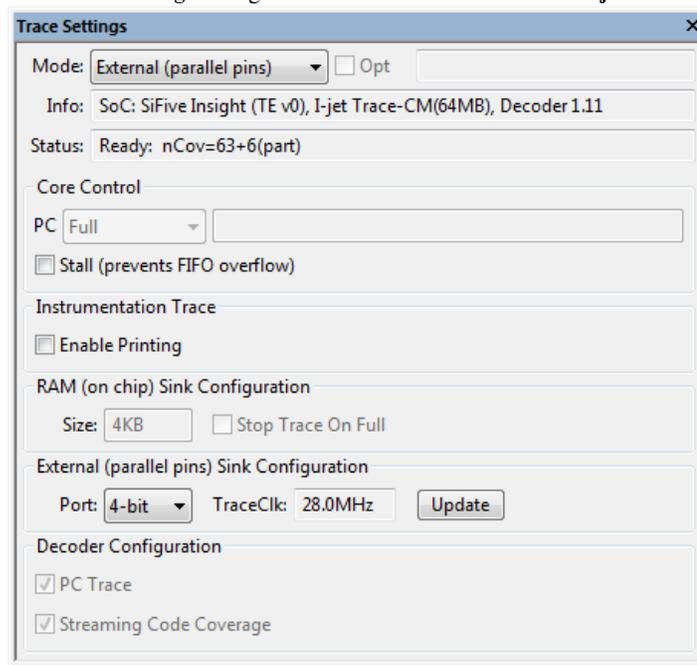


Click this button to open the **Trace Settings** dialog box, see *Trace Settings dialog box*, page 180. Hover over the button to get information about whether trace is on or off and

which functionality that is used. The button is green when trace is enabled and gray when it is disabled.

Trace Settings dialog box

The Trace Settings dialog box is available from the C-SPY I-jet menu.



Use this dialog box to configure trace generation and collection.

See also *Getting started with trace*, page 177.

Requirements

The I-jet driver and a device that supports trace.

Mode

The main trace mode. Some changes to the trace mode are not possible during a debug session; see the Debug Log for detailed information.

Note: Not all types of trace can be used by all combinations of debug probe and target hardware. A setting that cannot be used will be ignored; see the Debug Log for detailed information.

Choose between:

Off (disabled)

This disables I-jet trace completely. If the I-jet trace mode is **Off**, trace cannot be enabled or used in any trace-related windows for the I-jet driver. This setting ensures that trace modules will not do any reading/writing to the target system.

If, for some reason, trace seems to be available in a debug session when the device you are debugging does not support it, you can use this setting to turn the (non-functional) trace features off.

Auto (probe dependent)

Selects External, RAM, or Serial trace, in that order, depending on the best match of the capabilities of the debug probe and the trace components of the target system.

External (parallel pins)

Collects External trace data, see *External trace*, page 176.

RAM (on chip)

Collects RAM trace data, see *RAM trace*, page 176.

Serial (serial pin)

Collects Serial trace data, see *Serial trace*, page 176.

Opt

Use this field with device families from some SoC vendors to make settings required to use trace. For detailed information, see the document

`RISC-V-Trace-Control-Interface.adoc`, available on github.com.

Info

Gives information about the target device and the debug probe, showing the amount of probe memory and the version of the decoder that is used for processing trace. This field is read-only.

Status

Displays the status of current trace capture. This field is read-only. The elements that can be displayed in this field are:

status:

The current status of the trace collection, one of: **Ready**, **Streaming** (capturing full trace in real time), **Active** (capturing and buffering trace), and **Done**.

Unread xMB (y%)

If the decoding is slower than the streaming data rate, x is the amount of raw trace not yet read ($y\%$ of the probe trace memory). For RAM trace, or if PC Trace is disabled, this element might be omitted.

nInst=z

z is the number of instructions that have been decoded from the processed raw trace stream.

nCov=a+b(part)

a is the number of completely covered instructions and b the number of partially executed instructions.



The messages in the **Debug Log** window give a fuller description of the trace collection.

Stall

Stalls the core to prevent on-chip FIFO overflow. Note that when core is stalled, real-time behavior might be affected. Use this option only when you see trace overflows.

Enable printing

Makes it possible for the executing application to send text to the `stdout` and `stderr` streams by way of trace output, rather than by temporarily stopping at a breakpoint.

Using this option requires that the standard C library is configured with the option to direct the `stdout/stderr` streams **via Trace ITC**. To configure the library this way, choose **Project>Options>General Options>Library Configuration**.

Size

Displays the on-chip RAM size when the **Mode** is set to **RAM**. This field is read-only.

Stop trace on full

Stops RAM trace collection before the trace buffer becomes full.

Port

Specifies which trace port pins to use when the **Mode** is set to **External**. A setting that cannot be used will be ignored; see the Debug Log for detailed information.

TraceClk

The frequency of the trace clock (TraceClk) pin used for External trace, as measured by the debug probe.

Update

Refreshes the measurement of the trace clock frequency.

PC Trace

Shows whether PC Trace records are being produced or not. This check box is read-only—the production of PC Trace records is enabled/disabled in the individual debugger windows that make use of them.

Note: Without PC Trace, the decoding of the trace stream improves as coverage maps and counters can be updated live.

Streaming code coverage

Shows whether code coverage maps and counters are being produced or not. This check box is read-only—streaming code coverage is enabled/disabled in the individual debugger windows that make use of it.

Trace window

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

This window displays the collected trace data.

The content of the **Trace** window depends on the C-SPY driver you are using.

See also *Collecting and using trace data*, page 177.

Requirements

One of these alternatives:

- The C-SPY simulator
- The I-jet driver and a device that supports trace

Trace toolbar

The toolbar in the **Trace** window contains:

**Enable/Disable**

Enables and disables collecting and viewing trace data in this 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 179.

**Find**

Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 194.

**Save**

Displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.

**Edit Settings**

In the C-SPY simulator, this button is not enabled.

For the C-SPY I-jet driver, this button displays the **Trace Settings** dialog box, see *Trace Settings dialog box*, page 180.

**Progress bar**

When a large amount of trace data has been collected, there might be a delay before all of it has been processed and can be displayed. The progress bar reflects that processing.

Display area (in the C-SPY simulator)

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

Timestamp	Trace	Read Addr	Read Data	Write Addr	Write Data
0	80000000 lui sp, 0				
1	80000004 addi gp, gp, 0x106				
2	80000008 lui sp, 1				
3	8000000C addi sp, sp, 0x110				
4	80000010 lui a0, 0x80000				
5	80000014 addi a0, a0, 0x40				
6	80000018 csrri zero, mtve...				
7	8000001C csrrs zero, mtve...				
8	80000020 lui a0, 2				
9	80000024 csrrs zero, msta...				
10	80000028 csrrwi zero, fcsr...				
11	8000002C c.jal __low_level...				
12	8000023C c.li a0, 1				
13	8000023E c.ret				
14	8000002E beq a0, zero, ...				
15	80000032 c.jal __iar_data...				
16	8000015A c.addi16sp -0x10				

This area contains these columns for the C-SPY simulator:

The leftmost column contains identifying icons to simplify navigation within the buffer:

-  The yellow diamond indicates the trace execution point, marking when target execution has started.
-  The right green arrow indicates a call instruction.
-  The left green arrow indicates a return instruction.
-  The dark green bookmark indicates a navigation bookmark.
-  The red arrow indicates an interrupt.
-  The violet bar indicates the results of a search.

Timestamp

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.

Read Addr, Read Data, Write Addr, Write Data

These columns show reads and writes to memory.

A red-colored row indicates that the previous row and the red row are not consecutive. This means that there is a gap in the collected trace data, for example because trace data has been lost due to an overflow.

Display area (in the C-SPY I-jet driver)

This area displays a collected sequence of executed machine instructions and other trace data.

	Timestamp	Address	Trace	Exc...
	786	0x40400124	c.addi16sp 0x10	
<	787	0x40400126	c.ret	
>	788	0x4040002e	c.jal exit	
			exit:	
	789	0x404001b2	c.j _exit	
			_exit:	
	790	0x40400194	addi sp, sp, -0x10	
	791	0x40400198	sw ra, 0xC(sp)	
	792	0x4040019c	sw a0, 8(sp)	
	793	0x404001a0	lw a0, 8(sp)	
>	794	0x404001a4	c.jal __exit	
			__exit:	
	795	0x404000de	c.addi16sp -0x20	

This area contains these columns for the C-SPY I-jet driver:

The leftmost column contains identifying icons to simplify navigation within the buffer:

-  The yellow diamond indicates the trace execution point, marking when target execution has started.
-  The right green arrow indicates a call instruction.
-  The left green arrow indicates a return instruction.
-  The dark green bookmark indicates a navigation bookmark.
-  The red arrow indicates an interrupt.
-  The violet bar indicates the results of a search.

Timestamp

An internal I-jet index number.

Address

The address of the instruction associated with the trace frame.

Trace

The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.

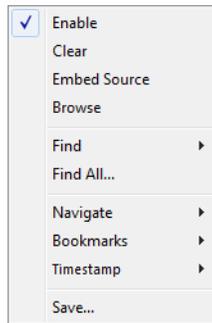
Except

The type of exception, when it occurs.

A red-colored row indicates that the previous row and the red row are not consecutive. This means that there is a gap in the collected trace data, for example because trace data has been lost due to an overflow.

Context menu

This context menu is available:



Note: The contents of this menu are dynamic and depend on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands. Note that the shortcuts to the submenu commands do not use the Ctrl key.

These commands are available:

Enable

Enables and disables collecting and viewing trace data in this window.

Clear

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

Embed 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 179.

Find>Find (F)

Displays a dialog box where you can perform a search in the **Trace** window, see *Find in Trace dialog box*, page 194. The contents of the window will scroll to display the first match.

Find>Find Next (G)

Finds the next occurrence of the specified string.

Find>Find Previous (Shift+G)

Finds the previous occurrence of the specified string.

Find>Clear (Shift+F)

Removes all search highlighting in the window.

Find All

Displays a dialog box where you can perform a search in the **Trace** window, see *Find in Trace dialog box*, page 194. The search results are displayed in the **Find in Trace** window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 195.

Navigate>After Current Loop (L)

Identifies the selected program counter and scans the trace data forward, collecting program counters, until it finds the same address again. It has now detected a loop. (Loops longer than 1000 instructions are not detected.) Then it navigates forward until it finds a program counter that is not part of the collected set. This is useful for navigating out of many iterations of an idle or polling loop.

Navigate>Before Current Loop (Shift+L)

Behaves as **After Current Loop**, but navigates backward out of the loop.

Navigate>After Current Function (U)

Navigates to the next unmatched return instruction. This is similar to stepping out of the current function.

Navigate>Before Current Function (Shift+U)

Navigates to the closest previous unmatched call instruction.

Navigate>Next Statement (S)

Navigates to the next instruction that belongs to a different C statement than the starting point. It skips function calls, i.e. it tries to reach the next statement in the starting frame.

Navigate>Previous Statement (Shift+S)

Behaves as **Next statement**, but navigates backward to the closest previous different C statement.

Navigate>Next on Same Address (A)

Navigates to the next instance of the starting program counter address, typically to the next iteration of a loop.

Navigate>Previous on Same Address (Shift+A)

Navigates to the closest previous instance of the starting program counter address.

Navigate>Next Interrupt (I)

Navigates to the next interrupt entry. (To then find the matching interrupt exit, follow up with **After Current Function**.)

Navigate>Previous Interrupt (Shift+I)

Navigates to the closest previous interrupt entry.

Navigate>Next Execution Start Point (E)

Navigates to the next point where the CPU was started, for example places where the application stopped at breakpoints, or was stepped.

Navigate>Previous Execution Start Point (Shift+E)

Navigates to the closest previous point where the CPU was started.

Navigate>Next Discontinuity (D)

Navigates to the next discontinuity in the trace data.

Navigate>Previous Discontinuity (Shift+D)

Navigates to the closest previous discontinuity in the trace data.

Bookmarks>Toggle (+)

Adds a new navigation bookmark or removes an existing bookmark.

Bookmarks>Goto Next (B)

Navigates to the next navigation bookmark.

Bookmarks>Goto Previous (Shift+B)

Navigates to the closest previous navigation bookmark.

Bookmarks>Clear All

Removes all navigation bookmarks.

Bookmarks>location (0-9)

At the bottom of the submenu, the ten most recently defined bookmarks are listed, with a shortcut key each from 0-9.

Timestamp>Set as Zero Point (Z)

Sets the selected row as a reference “zero” point in the collected sequence of trace data. The count of rows in the **Trace** window will show this row as 0 and recalculate the timestamps of all other rows in relation to this timestamp.

Timestamp>Go to Zero Point (Shift+Z)

Navigates to the reference “zero” point in the collected sequence of trace data (if you have set one).

Timestamp>Clear Zero Point

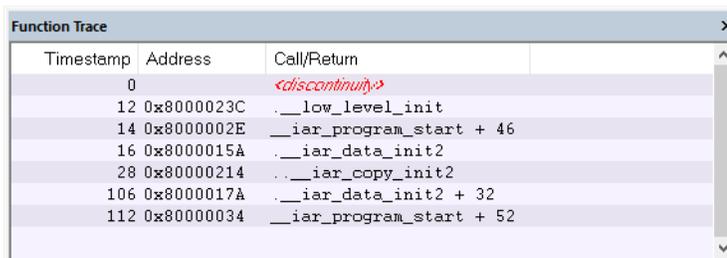
Removes the reference “zero” point from the trace data and restores the original timestamps of all rows.

Save

Displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.

Function Trace window

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



Timestamp	Address	Call/Return
0		<discontinuity>
12	0x8000023C	__low_level_init
14	0x8000002E	__iar_program_start + 46
16	0x8000015A	__iar_data_init2
28	0x80000214	__iar_copy_init2
106	0x8000017A	__iar_data_init2 + 32
112	0x80000034	__iar_program_start + 52

This window displays a subset of the trace data displayed in the **Trace** window. Instead of displaying all rows, the **Function Trace** window shows:

- The functions called or returned to, instead of the traced instruction
- The corresponding trace data.

Requirements

One of these alternatives:

- The C-SPY simulator
- The I-jet driver and a device that supports trace

Display area

For the C-SPY I-jet driver and the C-SPY simulator, and depending on the trace source, these columns are available:

Timestamp

For the simulator, this shows the number of cycles elapsed to this point.

For the I-jet driver, this shows an internal I-jet index number.

Address

The address of the executed instruction.

Call/Return

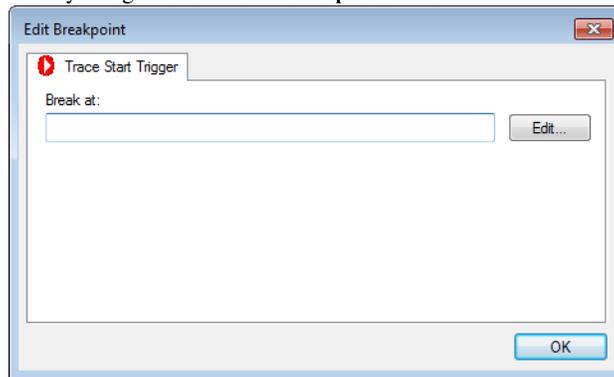
The function that was called or returned to.

Context menu

The context menu in this window is a subset of the context menu in the **Trace** window. All operations performed using this context menu will have effect also in the **Trace** window, and vice versa. For a description of the menu commands, see *Trace window*, page 183.

Trace Start Trigger breakpoint dialog box

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



Use this dialog box to set a Trace Start Trigger breakpoint where you want to start collecting trace data. If you want to collect trace data only for a specific range, you must also set a Trace Stop Trigger breakpoint where you want to stop collecting data.

See also *Trace Stop Trigger breakpoint dialog box*, page 193 and *Trace data collection using breakpoints*, page 178.

To set a Trace Start Trigger breakpoint:

- 1** In the editor or **Disassembly** window, right-click and choose **Trace Start Trigger** 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 Trigger**.
Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.
- 3** In the **Break at** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 4** When the breakpoint is triggered, the trace data collection starts.

Requirements

One of these alternatives:

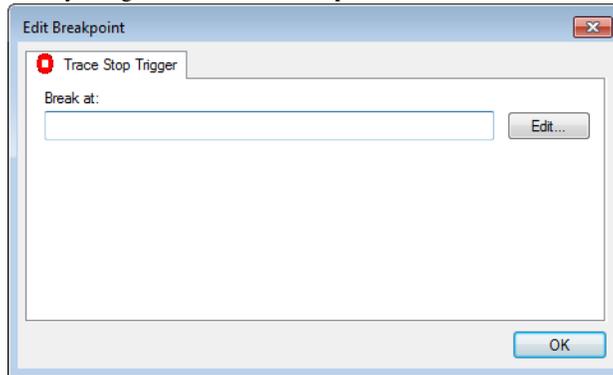
- The C-SPY simulator
- The I-jet driver and a device that supports trace

Break At

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Trace Stop Trigger breakpoint dialog box

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



Use this dialog box to set a Trace Stop Trigger breakpoint where you want to stop collecting trace data. If you want to collect trace data only for a specific range, you might also need to set a Trace Start Trigger breakpoint where you want to start collecting data.

See also *Trace Start Trigger breakpoint dialog box*, page 191 and *Trace data collection using breakpoints*, page 178.

To set a Trace Stop Trigger breakpoint:

- 1 In the editor or **Disassembly** window, right-click and choose **Trace Stop Trigger** 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 Trigger**.

Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.

- 3 In the **Break at** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 4 When the breakpoint is triggered, the trace data collection stops.

Requirements

One of these alternatives:

- The C-SPY simulator
- The I-jet driver and a device that supports trace

Break At

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 126.

Find in Trace dialog box

The **Find in Trace** dialog box is available by clicking the **Find** button on the **Trace** window toolbar or by choosing **Edit>Find and Replace>Find**.

Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the **Trace** window is the current window or the **Find** dialog box if the editor window is the current window.

Use this dialog box to specify the search criteria for advanced searches in the trace data.

The search results are displayed in the **Find in Trace** window—available from the **View>Messages** menu, see *Find in Trace window*, page 195.

See also *Searching in trace data*, page 178.

Requirements

One of these alternatives:

- The C-SPY simulator
- The I-jet driver and a device that supports trace

Text search

Specify the string you want to search for. To specify the search criteria, choose between:

Match case

Searches only for occurrences that exactly match the case of the specified text. Otherwise **int** will also find **INT**, **Int**, and so on.

Match whole word

Searches only for the string when it occurs as a separate word. Otherwise **int** will also find **print**, **sprintf**, and so on.

Only search in one column

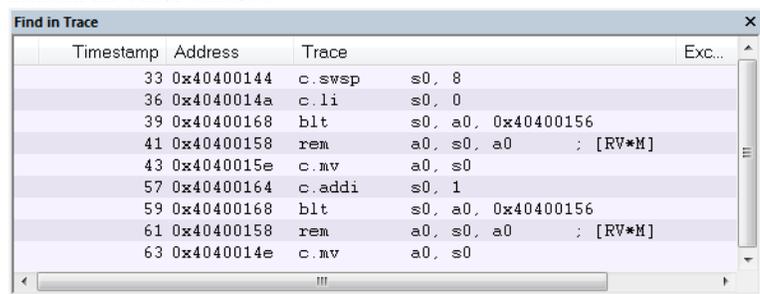
Searches only in the column you selected from the drop-down list.

Address range

Specify the address range you want to display or search. The trace data within the address range is displayed. If you have also 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.



Timestamp	Address	Trace
33	0x40400144	c.swsp s0, 8
36	0x4040014a	c.li s0, 0
39	0x40400168	blt s0, a0, 0x40400156
41	0x40400158	rem a0, s0, a0 ; [RV*M]
43	0x4040015e	c.mv a0, s0
57	0x40400164	c.addi s0, 1
59	0x40400168	blt s0, a0, 0x40400156
61	0x40400158	rem a0, s0, a0 ; [RV*M]
63	0x4040014e	c.mv a0, s0

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 194.

See also *Searching in trace data*, page 178.

Requirements

One of these alternatives:

- The C-SPY simulator
- The I-jet driver and a device that supports trace

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.

The application timeline

- Introduction to analyzing your application's timeline
- Analyzing your application's timeline
- Reference information on application timeline

Introduction to analyzing your application's timeline

These topics are covered:

- Briefly about analyzing the timeline
- Requirements for timeline support

See also:

- *Trace*, page 175

BRIEFLY ABOUT ANALYZING THE TIMELINE

C-SPY can provide information for various aspects of your application, collected when the application is running. This can help you to analyze the application's behavior.

You can view the timeline information in different representations:

- As different *graphs* that correlate with the running application in relation to a shared *time axis*.
- As detailed logs
- As summaries of the logs.

Timeline information can be provided for:

Call stack Can be represented in the **Timeline** window, as a graph that displays the sequence of function calls and returns collected by the trace system. You get timing information between the function invocations.

Note that there is also a related **Call Stack** window and a **Function Trace** window, see *Call Stack window*, page 70 and *Function Trace window*, page 190, respectively.

Data logging Based on data logs collected by the trace system for up to four different variables or address ranges, specified by means of *Data Log breakpoints*. Choose to display the data logs:

- In the **Timeline** window, as a graph of how the values change over time.
- In the **Data Log** window and the **Data Log Summary** window.

Interrupt logging Based on interrupt logs collected by the trace system. Choose to display the interrupt logs:

- In the **Timeline** window, as a graph of the interrupt events during the execution of your application.
- In the **Interrupt Log** window and the **Interrupt Log Summary** window.

Interrupt logging can, for example, help you locate which interrupts you can fine-tune to make your application more efficient.

For more information, see the chapter *Interrupts*.

REQUIREMENTS FOR TIMELINE SUPPORT

Depending on the capabilities of the hardware, the debug probe, and the C-SPY driver you are using, timeline information is supported for:

Target system	Call Stack	Data logging	Interrupt logging
C-SPY simulator	Yes	Yes	Yes
C-SPY I-Jet driver	Yes	—	—

Table 7: Support for timeline information

For more information about requirements related to trace data, see *Requirements for using trace*, page 177.

Analyzing your application's timeline

These tasks are covered:

- Displaying a graph in the Timeline window
- Navigating in the graphs
- Analyzing performance using the graph data
- Getting started using data logging

See also:

- *Using the interrupt system*, page 247

DISPLAYING A GRAPH IN THE TIMELINE WINDOW

The **Timeline** window can display several graphs—follow this example procedure to display any of these graphs. For an overview of the graphs and what they display, see *Briefly about analyzing the timeline*, page 197.

- 1 Choose **Timeline** from the C-SPY driver menu to open the **Timeline** window.
- 2 In the **Timeline** window, right-click in the window and choose **Select Graphs** from the context menu to select which graphs to be displayed.
- 3 In the **Timeline** window, right-click in the graph area and choose **Enable** from the context menu to enable a specific graph.
- 4 Click **Go** on the toolbar to start executing your application. The graphs that you have enabled appear.

NAVIGATING IN THE GRAPHS

After you have performed the steps in *Displaying a graph in the Timeline window*, page 199, you can use any of these alternatives to navigate in the graph:

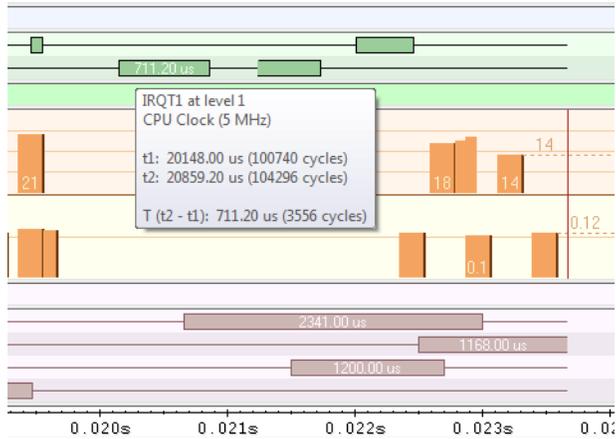
- Right-click and from the context menu choose **Zoom In** or **Zoom Out**. Alternatively, use the + and – keys. The graph zooms in or out depending on which command you used.
- Right-click in the graph and from the context menu choose **Navigate** and the appropriate command to move backwards and forwards on the graph. Alternatively, use any of the shortcut keys—arrow keys, Home, End, and Ctrl+End.
- Double-click on a sample of interest to highlight the corresponding source code in the editor window and in the **Disassembly** window.
- Click on the graph and drag to select a time interval, which will correlate to the running application. The selection extends vertically over all graphs, but appears highlighted in a darker color for the selected graph. Press Enter or right-click and from the context menu choose **Zoom>Zoom to Selection**. The selection zooms in. Use the navigation keys in combination with the Shift key to extend the selection.

ANALYZING PERFORMANCE USING THE GRAPH DATA

The **Timeline** window provides a set of tools for analyzing the graph data.

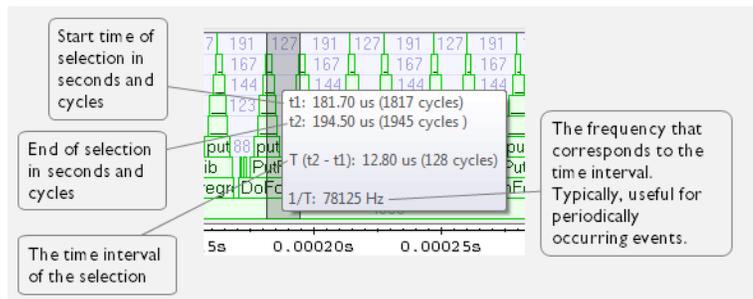
- 1 In the **Timeline** window, right-click and choose **Time Axis Unit** from the context menu. Select which unit to be used on the time axis—choose between **Seconds** and **Cycles**. If **Cycles** is not available, the graphs are based on different clock sources.

- 2 Execute your application to display a graph, following the steps described in *Displaying a graph in the Timeline window*, page 199.
- 3 Whenever execution stops, point at the graph with the mouse pointer to get detailed tooltip information for that location.



Note that if you have enabled several graphs, you can move the mouse pointer over the different graphs to get graph-specific information.

- 4 Click in the graph and drag to select a time interval. Point in the graph with the mouse pointer to get timing information for the selection.



GETTING STARTED USING DATA LOGGING

- 1 To set a data log breakpoint, use one of these methods:
 - In the **Breakpoints** window, right-click and choose **New Breakpoint>Data Log** to open the breakpoints dialog box. Set a breakpoint on the memory location that you want to collect log information for. This can be specified either as a variable or as an address.
 - In the **Memory** window, select a memory area, right-click and choose **Set Data Log Breakpoint** from the context menu. A breakpoint is set on the start address of the selection.
 - In the editor window, select a variable, right-click and choose **Set Data Log Breakpoint** from the context menu. The breakpoint will be set on the part of the variable that the microcontroller can access using one instruction.

You can set up to four data log breakpoints. For more information, see *Data Log breakpoints*, page 107.

- 2 Choose **C-SPY driver>Data Log** to open the **Data Log** window. Optionally, you can also choose:
 - **C-SPY driver>Data Log Summary** to open the **Data Log Summary** window
- 3 From the context menu, available in the **Data Log** window, choose **Enable** to enable the logging.
- 4 Start executing your application program to collect the log information.
- 5 To view the data log information, look in the **Data Log** window, or the **Data Log Summary** window.
- 6 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
- 7 To disable data logging, choose **Disable** from the context menu in each window where you have enabled it.

Reference information on application timeline

Reference information about:

- *Timeline window—Call Stack graph*, page 207
- *Data Log window*, page 202
- *Data Log Summary window*, page 205
- *Viewing Range dialog box*, page 212

See also:

- *Timeline window—Interrupt Log graph*, page 263

Data Log window

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

Time	Program Counter	l1	Address	s2	Address
0.160us	---			W 0x0000	@ 0x2004
0.160us	0xFFE00049	-	@ 0x2000		
24.480us	0xFFE000B5			R 0x0000	@ 0x2006
24.720us	0xFFE000BF			R 0x0042	@ 0x2004
24.760us	0xFFE000C6			R 0x0042	@ 0x2006
24.960us	0xFFE000E4	W 0x00004444	@ 0x2000		
<i>78.760us</i>	0xFFE00104			R 0x0042	@ 0x2004+?
79.000us	---			W 0x0084	@ 0x2004
100.800us	0xFFE00104			R 0x0084	@ 0x2006
101.040us	0xFFE0010E			W 0x00C6	@ 0x2004
<i>136.640us</i>	Overflow				
136.880us	0xFFE0010E			-	@ 0x2004

White rows indicate read accesses

Grey rows indicate write accesses

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

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

See also *Getting started using data logging*, page 201.

Requirements

The C-SPY simulator.

Display area

Each row in the display area shows the time, the program counter, and, for every tracked data object, its value and address. All information is cleared on reset. The information is displayed in these columns:

Time

The time for the data access is based on the clock frequency.

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

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

Cycles

The number of cycles from the start of the execution until the event.

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

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

Program Counter*

Displays one of these:

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

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

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

Value

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

To specify what data you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 107.

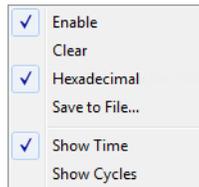
Address

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

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

Context menu

This context menu is available:



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 also happen when you reset the debugger.

Hexadecimal

Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.

Save to File

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

Show Time

Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.

Show Cycles

Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Data Log Summary window

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

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

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

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

See also *Getting started using data logging*, page 201.

Requirements

The C-SPY simulator.

Display area

Each row in this area displays the type and the number of accesses to each memory location or area in these columns. Summary information is listed at the bottom of the display area.

Data

The name of the data object you have selected to log accesses to. To specify what data object you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 107.

Total Accesses

The total number of accesses.

If the sum of read accesses and write accesses is less than the total accesses, the target system for some reason did not provide valid access type information for all accesses.

Read Accesses

The total number of read accesses.

Write Accesses

The total number of write accesses.

Unknown Accesses

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

Approximative time count

The information displayed depends on the C-SPY driver you are using.

For some C-SPY drivers, this information is not displayed or the value is always zero. In this case, all logs have an exact time stamp.

For other C-SPY drivers, a non-zero value is displayed. The value represents the amount of logs with an approximative time stamp. This might happen if the bandwidth in the communication channel is too low compared to the amount of data packets generated by the CPU or if the CPU generated packets with an approximative time stamp.

Overflow count

The information displayed depends on the C-SPY driver you are using.

For some C-SPY drivers, this information is not displayed or the value is always zero.

For other C-SPY drivers, the number represents the amount of overflows in the communication channel which can cause logs to be lost. If this happens, it indicates that logs might be incomplete. To solve this, make sure not to use all C-SPY log features simultaneously or check used bandwidth for the communication channel.

Current time/Current cycles

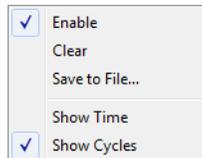
The information displayed depends on the C-SPY driver you are using.

For some C-SPY drivers, the value is always zero or not visible at all.

For other C-SPY drivers, the number represents the current time or cycles—the number of cycles or the execution time since the start of execution.

Context menu

This context menu is available:



This window displays trace data represented as different graphs, in relation to a shared time axis.

The Call Stack graph displays the sequence of function calls and returns collected by the trace system.

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

Requirements

One of these alternatives:

- The C-SPY simulator
- The I-jet driver and a device that supports trace

Display area for the Call Stack graph

Each function invocation is displayed as a horizontal bar which extends from the time of entry until the return. Called functions are displayed above its caller. The horizontal bars use four different colors:

- Medium green for normal C functions with debug information
- Light green for functions known to the debugger through an assembler label
- Medium yellow for normal interrupt handlers, with debug information
- Light yellow for interrupt handlers known to the debugger through an assembler label

The timing information represents the number of cycles spent in, or between, the function invocations.

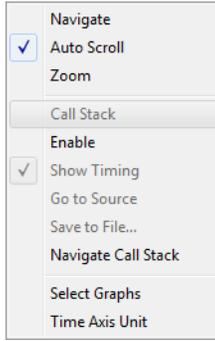
At the bottom of the window, there is a shared time axis that uses seconds or cycles as the time unit.

Click in the graph to display the corresponding source code.

Note: For highly optimized code, C-SPY might not be able to identify all calls. This means that for highly optimized code, the call stack is not entirely trustworthy.

Context menu

This context menu is available:



Note: The contents of this menu are dynamic and depend on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Navigate

Commands for navigating 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

Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.

Zoom

Commands for zooming the window, in other words, changing the time scale. Choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +

Zoom Out zooms out on the time scale. Shortcut key: –

10ns, 100ns, 1us, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

1ms, 10ms, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window.

10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Call Stack

A heading that shows that the Call stack-specific commands below are available.

Enable

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.

Show Timing

Toggles the display of the timing information on or off.

Go To Source

Displays the corresponding source code in an editor window, if applicable.

Save to File

Saves all contents (or the selected contents) of the Call Stack graph to a file. The menu command is only available when C-SPY is not running.

Navigate Call Stack>After Current Loop (L)

Identifies the selected program counter and scans the trace data forward, collecting program counters, until it finds the same address again. It has now detected a loop. (Loops longer than 1000 instructions are not detected.) Then it navigates forward until it finds a program counter that is not part of the collected set. This is useful for navigating out of many iterations of an idle or polling loop.

Navigate Call Stack>Before Current Loop (Shift+L)

Behaves as **After Current Loop**, but navigates backward out of the loop.

Navigate Call Stack>After Current Function (U)

Navigates to the next unmatched return instruction. This is similar to stepping out of the current function.

Navigate Call Stack>Before Current Function (Shift+U)

Navigates to the closest previous unmatched call instruction.

Navigate Call Stack>Next Statement (S)

Navigates to the next instruction that belongs to a different C statement than the starting point. It skips function calls, i.e. it tries to reach the next statement in the starting frame.

Navigate Call Stack>Previous Statement (Shift+S)

Behaves as **Next statement**, but navigates backward to the closest previous different C statement.

Navigate Call Stack>Next on Same Address (A)

Navigates to the next instance of the starting program counter address, typically to the next iteration of a loop.

Navigate Call Stack>Previous on Same Address (Shift+A)

Navigates to the closest previous instance of the starting program counter address.

Navigate Call Stack>Next Interrupt (I)

Navigates to the next interrupt entry. (To then find the matching interrupt exit, follow up with **After Current Function**.)

Navigate Call Stack>Previous Interrupt (Shift+I)

Navigates to the closest previous interrupt entry.

Navigate Call Stack>Next Execution Start Point (E)

Navigates to the next point where the CPU was started, for example places where the application stopped at breakpoints, or was stepped.

Navigate Call Stack>Previous Execution Start Point (Shift+E)

Navigates to the closest previous point where the CPU was started.

Navigate Call Stack>Next Discontinuity (D)

Navigates to the next discontinuity in the trace data.

Navigate Call Stack>Previous Discontinuity (Shift+D)

Navigates to the closest previous discontinuity in the trace data.

Select Graphs

Selects which graphs to be displayed in the **Timeline** window.

Time Axis Unit

Selects the unit used in the time axis—choose between **Seconds** and **Cycles**.

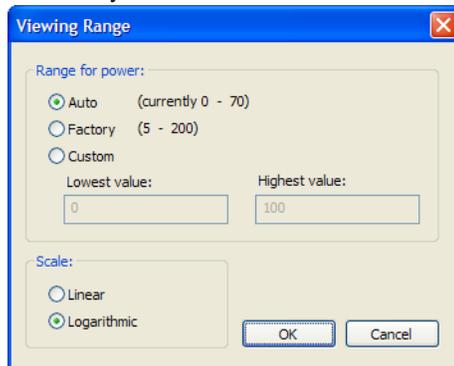
If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.

Profile Selection

Enables profiling time intervals in the **Function Profiler** window. Note that this command is only available if the C-SPY driver supports PC Sampling.

Viewing Range dialog box

The **Viewing Range** dialog box is available from the context menu that appears when you right-click in any graph in the **Timeline** window that uses the linear, levels or columns style.



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

Requirements

The C-SPY simulator.

Range for ...

Selects the viewing range for the displayed values:

Auto

Uses the range according to the range of the values that are actually collected, continuously keeping track of minimum or maximum values. The currently computed range, if any, is displayed in parentheses. The range is rounded to reasonably *even* limits.

Factory

For the Power Log graph—Uses the range according to the properties of the measuring hardware (only if supported by the product edition you are using).

For all other graphs—Uses the range according to the value range of the variable, for example 0–65535 for an unsigned 16-bit integer.

Custom

Use the text boxes to specify an explicit range.

Scale

Selects the scale type of the Y-axis:

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

Profiling

- Introduction to the profiler
- Using the profiler
- Reference information on the profiler

Introduction to the profiler

These topics are covered:

- Reasons for using the profiler
- Briefly about the profiler
- Requirements for using the profiler

REASONS FOR USING THE PROFILER

Function profiling can help you find the functions in your source code where the most time is spent during execution. You should focus on those functions when optimizing your code. A simple method of optimizing a function is to compile it using speed optimization. Alternatively, you can move the data used by the function into more efficient memory. For detailed information about efficient memory usage, see the *IAR C/C++ Development Guide for RISC-V*.

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

Instruction profiling can help you fine-tune your code on a very detailed level, especially for assembler source code. Instruction profiling can also help you to understand where your compiled C/C++ source code spends most of its time, and perhaps give insight into how to rewrite it for better performance.

BRIEFLY ABOUT THE PROFILER

Function profiling information is displayed in the **Function Profiler** window, that is, timing information for the functions in an application. Profiling must be turned on explicitly using a button on the window's toolbar, and will stay enabled until it is turned off.

For debug probes that support it, C-SPY can capture full instruction trace in real time, and process the information for the **Function Profiler** window.

Instruction profiling information is displayed in the **Disassembly** window, that is, the number of times each instruction has been executed.

Profiling sources

The profiler can use different mechanisms, or *sources*, to collect profiling information. Depending on the available trace source features, one or more of the sources can be used for profiling:

- *Trace (calls)*

The full instruction trace is analyzed to determine all function calls and returns.

When the collected instruction sequence is incomplete or discontinuous, the profiling information is less accurate.

- *Trace (flat)*

Each instruction in the full instruction trace or each PC Sample is assigned to a corresponding function or code fragment, without regard to function calls or returns.

This is most useful when the application does not exhibit normal call/return sequences, such as when you are using an RTOS, or when you are profiling code which does not have full debug information.

REQUIREMENTS FOR USING THE PROFILER

The C-SPY simulator supports the profiler; there are no specific requirements.

To use the profiler in a hardware debugger system, you need an I-jet or I-jet Trace debug probe and a device that supports trace. For information about the different types of trace data and their limitations, see *Briefly about trace*, page 175.

This table lists the C-SPY driver profiling support:

C-SPY driver	Trace (calls)	Trace (flat)
C-SPY simulator	Yes	Yes
C-SPY I-Jet driver	—	Yes
GDB Server	—	—

Table 8: C-SPY driver profiling support

Using the profiler

These tasks are covered:

- Getting started using the profiler on function level

- Analyzing the profiling data
- Getting started using the profiler on instruction level

GETTING STARTED USING THE PROFILER ON FUNCTION LEVEL

To display function profiling information in the **Function Profiler** window:

- 1 Build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Include debug information in output

Table 9: Project options for enabling the profiler



- 2 When you have built your application and started C-SPY, choose **C-SPY driver>Function Profiler** to open the **Function Profiler** window, and click the **Enable** button to turn on the profiler. Alternatively, choose **Enable** from the context menu that is available when you right-click in the **Function Profiler** window.
- 3 Start executing your application to collect the profiling information.
- 4 Profiling information is displayed in the **Function Profiler** window. To sort, click on the relevant column header.
- 5 When you start a new sampling, you can click the **Clear** button—alternatively, use the context menu—to clear the data.



ANALYZING THE PROFILING DATA

Here follow some examples of how to analyze the data.

The first figure shows the result of profiling using **Source: Trace (calls)**. The profiler follows the program flow and detects function entries and exits.

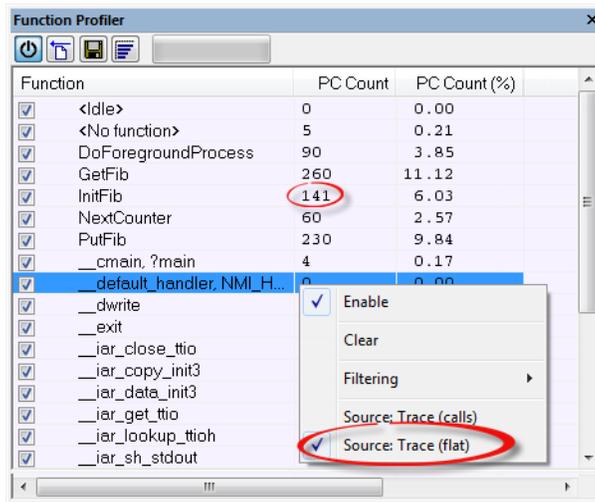
- For the **InitFib** function, **Flat Time** 231 is the time spent inside the function itself.
- For the **InitFib** function, **Acc Time** 487 is the time spent inside the function itself, including all functions **InitFib** calls.
- For the **InitFib/GetFib** function, **Acc Time** 256 is the time spent inside **GetFib** (but only when called from **InitFib**), including any functions **GetFib** calls.

- Further down in the data, you can find the **GetFib** function separately and see all of its subfunctions (in this case none).

Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
main	1	165	3.58	4356	94.39
DoForegroundProcess	10			3704	
InitFib	1			487	
PutFib	10	3174	68.78	3174	68.78
NextCounter	10	100	2.17	100	2.17
InitFib	1	231	5.01	487	10.55
GetFib	16			256	
GetFib	26	416	9.01	416	9.01
DoForegroundProcess	10	270	5.85	3704	80.26
GetFib	10			160	
NextCounter	10				
PutFib	10				
<Other>	0	25			98.85
main	1				

The second figure shows the result of profiling using **Source: Trace (flat)**. In this case, the profiler does not follow the program flow, instead the profiler only detects whether the PC address is within the function scope. For incomplete trace data, the data might contain minor errors.

For the `InitFib` function, **Flat Time** 231 is the time (number of hits) spent inside the function itself.



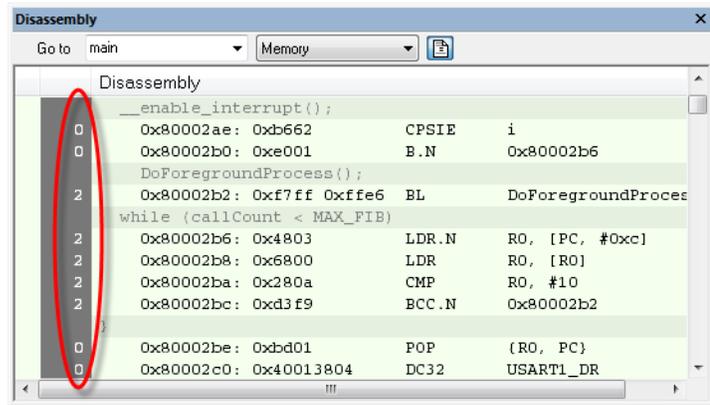
To secure valid data when using a debug probe, make sure to use the maximum trace buffer size and set a breakpoint in your code to stop the execution before the buffer is full.

Note: The `<No function>` entry represents PC values that are not within the known C-SPY ranges for the application.

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.



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

Reference information on the profiler

Reference information about:

- *Function Profiler window*, page 220

See also:

- *Disassembly window*, page 65
- *Trace Settings dialog box*, page 180

Function Profiler window

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

Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
DoForegroundProcess	49	5770	31.14	7198	38.84
GetFib	0	0	0.00	0	0.00
InitFib	0	0	0.00	0	0.00
InitUart	0	0	0.00	0	0.00
PutFib	4	1332	7.19	1332	7.19
UartReceiveHandler	4	96	0.52	1428	7.71
main	0	0	0.00	0	0.00

This window displays function profiling information.

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

See also *Using the profiler*, page 216.

Requirements

One of these alternatives:

- The C-SPY simulator
- The I-jet driver and a device that supports trace

Toolbar

The toolbar contains:



Enable/Disable

Enables or disables the profiler.



Clear

Clears all profiling data.



Save

Opens a standard **Save As** dialog box where you can save the contents of the window to a file, with tab-separated columns. Only non-expanded rows are included in the list file.



Graphical view

Overlays the values in the percentage columns with a graphical bar.

Progress bar

Displays a backlog of profiling data that is still being processed. If the rate of incoming data is higher than the rate of the profiler processing the data, a backlog is accumulated. The progress bar indicates that the profiler is still processing data, but also approximately how far the profiler has come in the process.

Note that because the profiler consumes data at a certain rate and the target system supplies data at another rate, the amount of data remaining to be processed can both increase and decrease. The progress bar can grow and shrink accordingly.

Display area

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

- *For the Trace (calls) source*, the display area contains one line for each function compiled with debug information enabled. When some profiling information has been collected, it is possible to expand rows of functions that have called other functions. The child items for a given function list all the functions that have been called by the parent function and the corresponding statistics.
- *For the Trace (flat) source*, the display area contains one line for each C function of your application, but also lines for sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels. Each executed PC address from trace data is treated as a separate sample and is associated with the corresponding line in the **Profiling** window. Each line contains a count of those samples.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 216.

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

Function (All sources)

The name of the profiled C function.

Calls (Trace (calls))

The number of times the function has been called.

Flat time (Trace (calls))

The time expressed as the estimated number of executed instructions spent inside the function.

Flat time (%) (Trace (calls))

Flat time expressed as a percentage of the total time.

Acc. time (Trace (calls))

The time expressed as the estimated number of executed instructions spent inside the function.

Acc. time (%) (Trace (calls))

Accumulated time expressed as a percentage of the total time.

PC Count (Trace (flat))

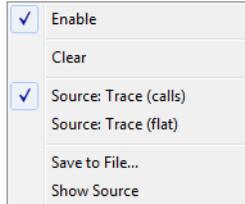
The number of executed instructions associated with the function.

PC Count (%) (Trace (flat))

The number of executed instructions associated with the function as a percentage of the total number of executed instructions.

Context menu

This context menu is available:



The contents of this menu depend on the C-SPY driver you are using.

These commands are available:

Enable

Enables the profiler. The system will also collect information when the window is closed.

Clear

Clears all profiling data.

Filtering

Selects which part of your code to profile. Choose between:

Check All—Excludes all lines from the profiling.

Uncheck All—Includes all lines in the profiling.

Load—Reads all excluded lines from a saved file.

Save—Saves all excluded lines to a file. Typically, this can be useful if you are a group of engineers and want to share sets of exclusions.

These commands are only available when using Trace (flat).

Source

Selects which source to be used for the profiling information. See also *Profiling sources*, page 216.

Note that the available sources depend on the C-SPY driver you are using.

Choose between:

Trace (calls)—the instruction count for instruction profiling is only as complete as the collected trace data.

Trace (flat)—the instruction count for instruction profiling is only as complete as the collected trace data.

Save to File

Saves all profiling data to a file.

Show Source

Opens the editor window (if not already opened) and highlights the selected source line.

Code coverage

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

Introduction to code coverage

These topics are covered:

- Reasons for using code coverage
- Briefly about code coverage
- Requirements and restrictions for using code coverage

REASONS FOR USING CODE COVERAGE

The code coverage functionality is useful when you design your test procedure to verify whether all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

BRIEFLY ABOUT CODE COVERAGE

The **Code Coverage** window reports the status of the current code coverage analysis for C or C++ code. 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.

For debug probes that support it, C-SPY can capture full instruction trace in real time, and process the information for the **Code Coverage** window.

Note: Assembler code is not covered in the **Code Coverage** window. To view code coverage for assembler code, use the **Disassembly** window.

REQUIREMENTS AND RESTRICTIONS FOR USING CODE COVERAGE

Code coverage is supported by the C-SPY simulator and there are no specific requirements or restrictions.

To use code coverage in a hardware debugger system, you need an I-jet or I-jet Trace debug probe and a device that supports trace. For information about the different types of trace data and their limitations, see *Briefly about trace*, page 175.

Using code coverage

These tasks are covered:

- Getting started using code coverage

GETTING STARTED USING CODE COVERAGE

To get started using code coverage:

- 1 Before you can use the code coverage functionality, you must build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Include debug information in output

Table 10: Project options for enabling code coverage

- 2 After you have built your application and started C-SPY, choose **View>Code Coverage** to open the **Code Coverage** window.
- 3  Click the **Activate** button, alternatively choose **Activate** from the context menu, to switch on code coverage.
- 4 Start the execution. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, the code coverage information is updated automatically.

Reference information on code coverage

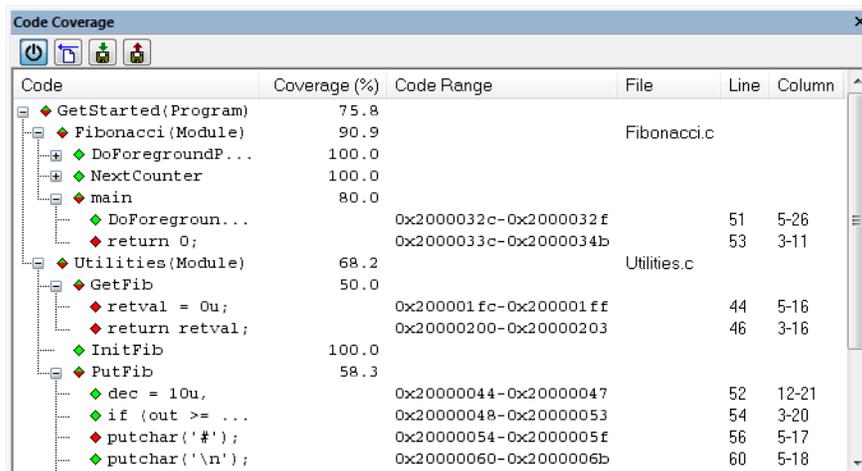
Reference information about:

- *Code Coverage window*, page 227

See also *Single stepping*, page 58.

Code Coverage window

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



Code	Coverage (%)	Code Range	File	Line	Column
◆ GetStarted(Program)	75.8				
◆ Fibonacci(Module)	90.9		Fibonacci.c		
◆ DoForegroundP...	100.0				
◆ NextCounter	100.0				
◆ main	80.0				
◆ DoForegroun...		0x2000032c-0x2000032f		51	5-26
◆ return 0;		0x2000033c-0x2000034b		53	3-11
◆ Utilities(Module)	68.2		Utilities.c		
◆ GetFib	50.0				
◆ retval = 0u;		0x200001fc-0x200001ff		44	5-16
◆ return retval;		0x20000200-0x20000203		46	3-16
◆ InitFib	100.0				
◆ PutFib	58.3				
◆ dec = 10u,		0x20000044-0x20000047		52	12-21
◆ if (out >= ...		0x20000048-0x20000053		54	3-20
◆ putchar('#');		0x20000054-0x2000005f		56	5-17
◆ putchar('\n');		0x20000060-0x2000006b		60	5-18

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.

Only source code that was compiled with debug information is displayed. Therefore, startup code, exit code, and library code are 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.

A statement is considered to be executed when all its instructions have been executed. By default, when a statement has been executed, it is removed from the window and the percentage is increased correspondingly.

Requirements

One of these alternatives:

- The C-SPY simulator
- The I-jet driver and a device that supports trace

Toolbar

The toolbar contains buttons for switching code coverage on and off, clearing the code coverage information, and saving/restoring the code coverage session. See the description of the context menu for more detailed information.

The toolbar contains these buttons:



Activate

Switches code coverage on and off during execution.



Clear

Clears the code coverage information. All step points are marked as not executed.



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 might not be supported by the C-SPY driver you are using.



Restore session

Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command might not be supported by the C-SPY driver you are using.

Display area

Double-clicking a statement or a function in the **Code Coverage** window displays that statement or function as the current position in the editor window, which becomes the active window.

These columns are available:

Code

The code coverage information is displayed as a tree structure, showing the program, module, function, and statement levels. You can use the plus (+) sign and minus (-) sign icons to expand and collapse the structure.

These icons give you an overview of the current status on all levels:

- Red diamond—0% of the modules or functions has been executed.
- Green diamond—100% of the modules or functions has been executed.
- Red and green diamond—Some of the modules or functions have been executed. This is most directly visible at the instruction level for a branch instruction or a conditionally executed instruction. For example, coverage

data can indicate that while a branch instruction has executed, control flow has not both branched and continued in straight execution. Or, in the case of a conditional instruction, it indicates if the instruction has executed both with the controlling flag set, and with the controlling flag not set.

Red, green, and yellow colors can be used as highlight colors in the source editor window. In the editor window, the yellow color signifies partially executed.

Coverage (%)

The amount of statements that has been covered so far, that is, the number of executed statements divided with the total number of statements.

Code Range

The address range in code memory where the statement is located.

File

The source file where the step point is located.

Line

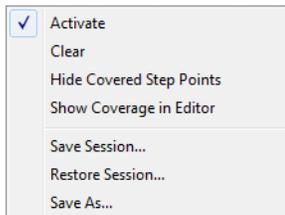
The source file line where the step point is located.

Column

The source file column where the step point is located.

Context menu

This context menu is available:



These commands are available:



Activate

Switches code coverage on and off during execution.



Clear

Clears the code coverage information. All step points are marked as not executed.

Hide Covered Step Points

Toggles the display of covered step points on and off. When this option is selected, executed statements are removed from the window.

Show Coverage in Editor

Toggles the red, green, and yellow highlight colors that indicate code coverage in the source editor window on and off.



Save session

Saves your code coverage session data to a * .dat file. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command might not be supported by the C-SPY driver you are using.



Restore session

Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command might not be supported by the C-SPY driver you are using.

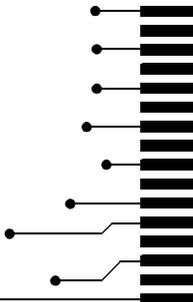
Save As

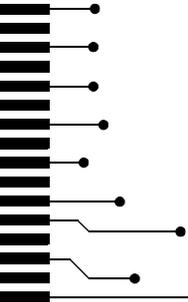
Saves the current code coverage result in a text file.

Part 3. Advanced debugging

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

- Multicore debugging
- Interrupts
- C-SPY macros
- The C-SPY command line utility—`cspybat`
- Flash loaders





Multicore debugging

- Introduction to multicore debugging
- Debugging multiple cores
- Reference information on multicore debugging

Introduction to multicore debugging

These topics are covered:

- Briefly about multicore debugging
- Symmetric multicore debugging
- Asymmetric multicore debugging
- Requirements and restrictions for multicore debugging

BRIEFLY ABOUT MULTICORE DEBUGGING

Multicore debugging means that you can debug targets with multiple cores. The C-SPY debugger supports multicore debugging in two ways:

- *Symmetric multicore debugging (SMP)*, which means debugging two or more identical cores. This is handled using a single instance of the IAR Embedded Workbench IDE.
- *Asymmetric multicore debugging (AMP)*, which means debugging two or more cores based on different architectures. It could be two different RISC-V-cores, for example an RV32EM and an RV32IMFD. This is handled using two or more cooperating instances of the IAR Embedded Workbench IDE.

SYMMETRIC MULTICORE DEBUGGING

Symmetric multicore debugging means that the target has two or more identical cores on the board (usually on the same chip) that typically can be accessed through a single debug probe.

In the debugger, at any given time the windows show the state of only one of the cores—the one in focus.

This is an overview of special support for symmetric multicore debugging:

- You can control which core you want the debugger to focus on. This affects editor windows and the **Disassembly**, **Registers**, **Watch**, **Locals**, **Call Stack** window, etc.

- The **Cores** window shows a list of all available cores, and gives some information about each core, such as its execution state. The **Multicore** toolbar is a complement to the **Cores** window,
- The **Stack** window can show the stack for each core by means of dedicated stack sections.
- RTOS support is available in separate multicore-aware plugins. Typically, they work like their single-core plugin counterparts, but handle multiple active tasks on separate cores. The plugins might also provide the information required by the **Stack** window to display the stack for any selected task.

ASYMMETRIC MULTICORE DEBUGGING

Asymmetric multicore means that the target has two or more cores based on different architectures. To debug the target, two or more IDE instances can be used, where each instance is connected to one or more identical cores. The IDE instances synchronize so that debugging sessions can be started and stopped, and the cores can be controlled from any of the instances. Except for shared memory, each debugging session can only show information (variables, call stack, etc) about its own cores.

You start one IDE instance manually and that instance is referred to as the *master*. When you start an asymmetric multicore debugging session, the *master* instance can initiate one or more *partner* (or *slave*) instances. The partner instances will be reused if they are already running.

All instances each require their own project, master and partners. You must set up each project with the correct processor variant, linker, and debugger options. The master project must also be configured to act as multicore master or have multicore master mode enabled.

One possible strategy for download is to combine the debug images for the cores into one and let the master project download the combined image. In this scenario, the partners must be configured to attach to a running target, and/or to suppress any downloading.

Another strategy is to download the master and partners as separate binary images, in which case you must make sure to avoid any unintentional overlaps in memory.

This is an overview of special support for asymmetric multicore debugging:

- You can control whether to automatically start and stop the whole application or to run the cores independently of each other.
- Each instance of the IDE displays debug information for the cores that it is connected to.

- The **Cores** window shows a list of all available cores, and gives some information about each core, such as its execution state. The **Multicore** toolbar is a complement to the **Cores** window,
- When you set a breakpoint, it is only connected to one core, and when the breakpoint is triggered, that core is stopped.

Note: Stepwise execution in an asymmetric multicore project has one limitation: When you step in the **Disassembly** window, or in a *C/C++* source window and no breakpoints are available, the stepping will only affect the current core.

REQUIREMENTS AND RESTRICTIONS FOR MULTICORE DEBUGGING

The C-SPY simulator supports multicore debugging and there are no specific requirements or restrictions.

To use multicore debugging in your hardware debugger system, you need a specific combination of C-SPY driver and debug probe:

- The IAR C-SPY I-jet driver
- An I-jet or I-jet Trace debug probe

Note: There might be restrictions in trace support due to limitations in the hardware you are using.

Debugging multiple cores

These tasks are covered:

- Setting up for symmetric multicore debugging
- Setting up for asymmetric multicore debugging
- Starting and stopping a multicore debug session

SETTING UP FOR SYMMETRIC MULTICORE DEBUGGING

- 1 Choose **Project>Options>Debugger>Multicore** and specify the number of cores you have.
- 2 You can now start your debug session.

SETTING UP FOR ASYMMETRIC MULTICORE DEBUGGING

There are a number of ways that you can set up for multicore debugging, but this strategy is recommended:

1: Preparing the projects

- 1 Create a workspace with two or more projects, one for each core or set of cores.
- 2 Select an appropriate download strategy. One possible download strategy would be to combine the images for the cores into one and let the master project download the combined image. In this scenario, the partners would have to be configured to attach to a running target, and/or to suppress any downloading.
- 3 In the intended master project, choose **Project>Options>Debugger>Multicore** to open the **Multicore** options page.

2A: Setting up one partner project

- 1 Select **Simple**. Specify the options **Partner workspace** (path), **Partner project** (project name), and **Partner configuration** (build configuration). These settings are used when the partner session starts.

- 2 Select the option **Attach partner to running target**. Use the **Partner cores** option to specify the number of cores in the partner project.

By default, the Embedded Workbench instance associated with the *partner* project must be installed in the same directory as the Embedded Workbench instance associated with the *master* project, for example in `c:\Program Files\IAR Systems\Embedded Workbench N.n`. If the two Embedded Workbench instances were installed in *different* locations (perhaps because they are not based on the same version (*N.n*) of the Embedded Workbench shared components), you must select **Override partner debugger location** and specify the installation directory of the Embedded Workbench for the partner project. Note that the Embedded Workbench for the partner project must

be based on version 9.1.7 or later of the shared components—to check this, choose **Help>About>Product Info**.

For more information about the multicore settings, see *Multicore*, page 377

2B: Setting up two or more partner projects

- 1 Select **Advanced**. Create a multicore session file in XML format with information about location and settings of the partner projects and use the browse button to specify this file. For more information about this file, see *The multicore session file*, page 241. These settings are used when the partner session starts.

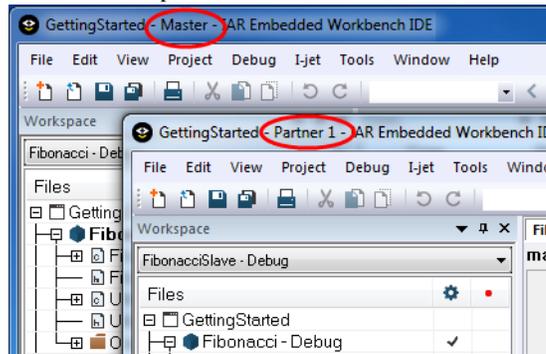
The image shows a configuration window titled "Multicore". It is divided into two sections: "Symmetric multicore" and "Asymmetric multicore".

- Symmetric multicore:**
 - Number of cores: 1
- Asymmetric multicore:**
 - Radio buttons: Disabled, Simple
 - Partner workspace: []
 - Partner project: []
 - Partner configuration: []
 - Attach partner to running target Partner cores: 2
 - Override partner debugger location
 - Partner debugger: []
 - Advanced
 - Session configuration: C:\Temp\SessionSpec.xml

3: Make final settings

- 1 Select appropriate reset strategies for all projects:
 - In the master project, choose **Project>Options>C-SPY driver>Setup>Reset** and select a reset strategy, typically **Hardware**.
 - In the **Workspace** window, switch to the partner projects one at a time. Then for each project, choose **Project>Options>C-SPY driver>Setup>Reset** and select a reset strategy for the partner session that does not affect the master session, typically **Software**.
- 2 Make sure to use compatible settings for the debug probe for all projects.

The master and partner instances are indicated in the main IDE window title bar.



STARTING AND STOPPING A MULTICORE DEBUG SESSION

- 1 To start a multicore debug session, for example use the standard **Download and Debug** command, either in the master or in a partner session.
- 2 To stop a multicore debug session, for example use the standard **Stop Debugging** command, which will stop all debugging sessions.

Reference information on multicore debugging

Reference information about:

- *Cores window*, page 239
- *Multicore toolbar*, page 241
- *The multicore session file*, page 241

See also:

- `__getNumberOfCores`, page 295
- `__getSelectedCore`, page 295
- `__selectCore`, page 308

Cores window

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

Core	Status	PC	Cycles
0: PE	Stopped	0x40400152	-
1: PE	Stopped	0x4040014C	-

This window shows a list of all available cores, and gives some information about each core, such as its execution state. The line highlighted in bold is the core currently in focus, which means that any window showing information that is specific to a core will be updated to reflect the state of the core in focus. This includes highlights in editor windows and the **Disassembly**, **Registers**, **Watch**, **Locals**, **Call Stack** window, and so on. Double-click a line to focus on that core.

Note: For asymmetric multicore debugging, only local cores can be in focus.

See also *Debugging multiple cores*, page 235.

Requirements

One of these alternatives:

- The C-SPY simulator
- An I-jet or I-jet Trace debug probe

Display area

A row in this area shows information about a core, in these columns:

Execution state

Displays one of these icons to indicate the execution state of the core:

	in focus, not executing
	not in focus, not executing
	in focus, executing
	not in focus, executing
	in focus, unknown status
	not in focus, unknown status

Core

The name of the core.

Status

The status of the execution, which can be one of **Stopped**, **Running**, **Sleeping**, or **Unknown**.

PC

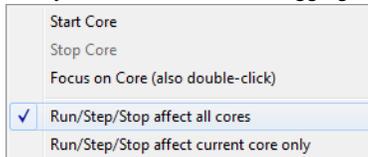
The value of the program counter.

Cycles | Time

The value of the cycle counter or the execution time since the start of the execution, depending on the debugger driver you are using.

Context menu

For symmetric multicore debugging, this context menu is available:



These commands are available:

Start Core

Starts the selected core.

Stop Core

Stops the selected core.

Focus on Core (also double-click)

Focuses on the selected core.

Run/Step/Stop affect all cores

The **Run**, **Step**, **Stop** commands affect all cores.

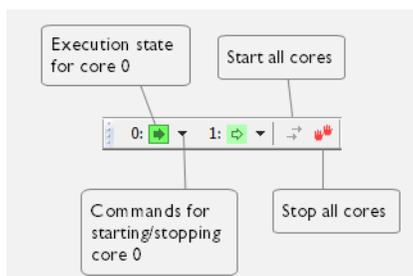
Run/Step/Stop affect current core only

The **Run/Step/Stop** commands only affect the current core. This menu command is only supported if your device supports it.

Note: These commands are not supported by all target hardware.

Multicore toolbar

The **Multicore** toolbar can be toggled on/off from the **Window>Toolbars** submenu when you have enabled multicore debugging, see *Setting up for asymmetric multicore debugging*, page 235.



This toolbar is a complement to and shows the same state as the **Cores** window. Each core has a button with an adjacent drop-down menu. Click a button to make C-SPY focus on that core.

Note: You can use the toolbar commands to start and stop cores in the associated debugging session.

The multicore session file

This file in XML format can be used to specify an asymmetric debug session with more than two IDE instances. You specify the location of the file to the IDE on the **Project>Options>Debugger>Multicore** page. For more information, see *Setting up for asymmetric multicore debugging*, page 235, and *Multicore*, page 377,

XML specification

The multicore session file needs to look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<sessionSetup>
  <partner>
    <name>Name_of_master_instance</name>
    <workspace>Path_to_workspace</workspace>
    <project>Name_of_project</project>
    <config>Build_config</config>
    <numberOfCores>N</numberOfCores>
    <attachToRunningTarget>true/false</attachToRunningTarget>
  </partner>
</sessionSetup>
```

```
<partner>
  <name>Name_of_partner_instance_1</name>
  <workspace>Path_to_workspace</workspace>
  <project>Name_of_project</project>
  <config>Build_config</config>
  <numberOfCores>N</numberOfCores>
  <attachToRunningTarget>true/false</attachToRunningTarget>

  <debuggerpath>Path_to_Embedded_workbench</debuggerpath>
</partner>

...

<partner>
  <name>Name_of_partner_instance_N</name>
  <workspace>Path_to_workspace</workspace>
  <project>Name_of_project</project>
  <config>Build_config</config>
  <numberOfCores>N</numberOfCores>
  <attachToRunningTarget>true/false</attachToRunningTarget>

  <debuggerpath>Path_to_Embedded_workbench</debuggerpath>
</partner>

</sessionSetup>
```

Interrupts

- Introduction to interrupts
- Using the interrupt system
- Reference information on interrupts

Introduction to interrupts

These topics are covered:

- Briefly about the interrupt simulation system
- Interrupt characteristics
- 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 283
- *Breakpoints*, page 105
- *The IAR C/C++ Development Guide for RISC-V*

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 RISC-V
- 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.

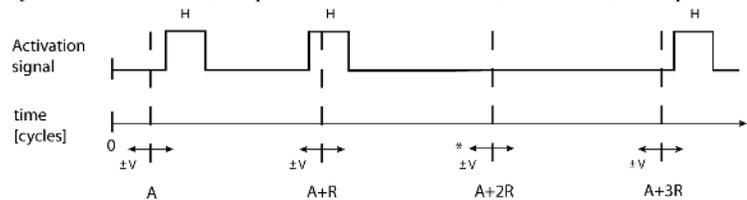
All interrupts you define using the **Interrupt Configuration** window are preserved between debug sessions, unless you remove them. A forced interrupt, on the other hand, exists only until it has been serviced and is not preserved between sessions.



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 Configuration** window or a system macro.

INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability*.



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

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

The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation time. However, interrupts can only be raised between instructions, which means that a full assembler instruction must have been executed before the interrupt is generated, regardless of how many cycles an instruction takes.

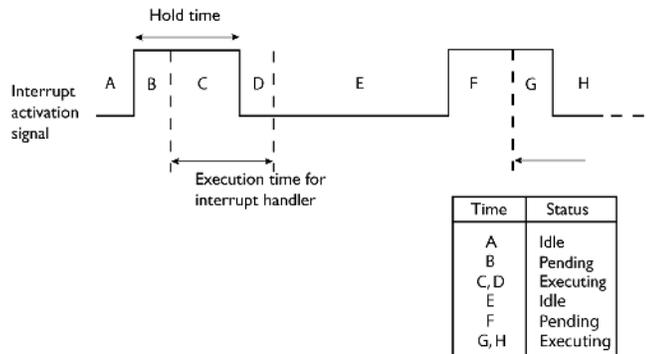
To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—

the probability, in percent, that the interrupt will actually appear in a period—and *variance*—a time variation range as a percentage of the repeat interval. These options make it possible to randomize the interrupt simulation. You can also specify a *hold time* which describes how long the interrupt remains pending until removed if it has not been processed. If the hold time is set to *infinite*, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

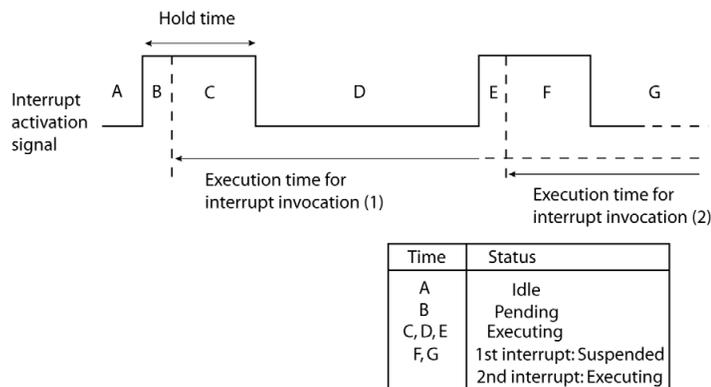
INTERRUPT SIMULATION STATES

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:



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:



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:

- `__cancelAllInterrupts`
- `__cancelInterrupt`
- `__disableInterrupts`
- `__enableInterrupts`
- `__orderInterrupt`
- `__popSimulatorInterruptExecutingStack`

The parameters of the first five macros correspond to the equivalent entries of the **Interrupt Configuration** window.

For more information about each macro, see *Reference information on C-SPY system macros*, page 283.

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 simulate device-specific interrupts, the interrupt system must have detailed information about each available interrupt. This information is provided in the device description files.

For information about device description files, see *Selecting a device description file*, page 43.

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. In the IDE:

- The logs are displayed in the **Interrupt Log** window
- 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

Requirements for interrupt logging

Interrupt logging is supported by the C-SPY simulator.

See also *Getting started using interrupt logging*, page 250.

Using the interrupt system

These tasks are covered:

- Simulating a simple interrupt
- Simulating an interrupt in a multi-task system
- Getting started using interrupt logging

See also:

- *Using C-SPY macros*, page 269 for details about how to use a setup file to define simulated interrupts at C-SPY startup
- The tutorial *Simulating an interrupt* in the Information Center.

SIMULATING A SIMPLE INTERRUPT

This example demonstrates the method for simulating a timer interrupt. However, the procedure can also be used for other types of interrupts.

To simulate and debug an interrupt:

- I Assume this simple application for a SiFive E24 Arty 100T device. It contains an interrupt service routine for a timer, which increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```
#pragma language = extended
#include <sdtio.h>
#include <stint.h>
#include <intrinsics.h>
#include <SiFive/ioe24arty.h>

static volatile uint_fast8_t ticks = 0;
void main (void)
{
    // Enable timer interrupt (CLINT)
    __set_bits_csr(_CSR_MIE, 0x80);

    __enable_interrupt();          /* Enable interrupts */

    while (ticks < 100);          /* Wait loop */

    // Disable timer interrupt (CLINT)
    __clear_bits_csr(_CSR_MIE, 0x80);
    __disable_interrupt(); /* Disable interrupts */
    printf("Done\n");
}

/* Timer interrupt service routine */
#pragma vector = 7
__interrupt void basic_timer(void)
{
    ticks += 1;
}
```

- 2 Add your interrupt service routine to your application source code and add the file to your project.
- 3 Choose **Project>Options>Debugger>Setup** and select a device description file. The device description file contains information about the interrupt that C-SPY needs to be able to simulate it. Use the **Use device description file** browse button to locate the `adaf` file.
- 4 Build your project and start the simulator.
- 5 Choose **Simulator>Interrupt Configuration** to open the **Interrupt Configuration** window. Right-click in the window and select **Enable Interrupt Simulation** on the context menu. For the timer example, verify these settings:

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

Table 11: Timer interrupt settings

Click **OK**.

- 6 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:
 - Generate an interrupt when the cycle counter has passed 4000
 - Continuously repeat the interrupt after approximately 2000 cycles.
- 7 To watch the interrupt in action, choose **Simulator>Interrupt Log** to open the **Interrupt Log** window.
- 8 From the context menu, available in the **Interrupt Log** window, choose **Enable** to enable the logging. If you restart program execution, status information about entrances and exits to and from interrupts will now appear in the **Interrupt Log** window.

For information about how to get a graphical representation of the interrupts correlated with a time axis, see *The application timeline*, page 197.

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 Configuration** window 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

- 1 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
- 2 From the context menu in the **Interrupt Log** window, choose **Enable** to enable the logging.
- 3 Start executing your application program to collect the log information.
- 4 To view the interrupt log information, look in the **Interrupt Log** or **Interrupt Log Summary** window, or the Interrupt graph in the **Timeline** window.
- 5 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.
- 6 To disable interrupt logging, from the context menu in the **Interrupt Log** window, toggle **Enable** off.

Reference information on interrupts

Reference information about:

- *Interrupt Configuration window*, page 251
- *Available Interrupts window*, page 254
- *Interrupt Status window*, page 255
- *Interrupt Log window*, page 257
- *Interrupt Log Summary window*, page 260

- *Timeline window—Interrupt Log graph, page 263*

Interrupt Configuration window

The **Interrupt Configuration** window is available by choosing **Simulator>Interrupt Configuration**.

Interrupt	Id	Type	Description	First Activation	Repeat Interval	Hold Time	Variance (%)	Probability (%)
<input checked="" type="checkbox"/> NMI	0	Single	-2 0x08	0	0	inf	0	100
<input checked="" type="checkbox"/> IEL0	1	Single	1 0x40	0	0	inf	0	100
<input checked="" type="checkbox"/> HardFault	2	Single	-1 0x0C PRI...	0	0	inf	0	100
<input type="checkbox"/> <click to add>								

Build Debug Log Interrupt Configuration

This window lists all installed interrupts. Use this window to enable or disable individual interrupts or the interrupt simulation system, and to edit the properties of installed interrupts.

See also *Using the interrupt system, page 247*.

Requirements

The C-SPY simulator.

Display area

This area contains these columns:

Interrupt

Lists all installed interrupts. Use the checkbox to enable or disable the interrupt.

ID

A unique interrupt identifier.

Type

Shows the type of the interrupt. The type can be one of:

Forced, a single-occasion interrupt defined in the **Available Interrupts** window.

Single, a single-occasion interrupt.

Repeat, a periodically occurring interrupt.

If the interrupt has been set from a C-SPY macro, the additional part (**macro**) is added, for example: **Repeat(macro)**.

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 vector address, the enable bit, the pending bit, the default priority, whether it is an exception or an interrupt, maskable or non-maskable, and the interrupt type (*Reserved*), separated by space characters. For interrupts specified using the system macro `__orderInterrupt`, the **Description** box is empty.

First Activation

The value of the cycle counter after which the specified interrupt will be generated. Click to edit.

Repeat Interval

The periodicity of the interrupt in cycles. Click to edit.

Hold Time

How long, in cycles, the interrupt remains pending until removed if it has not been processed. Click to edit. If you specify `inf`, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

Variance %

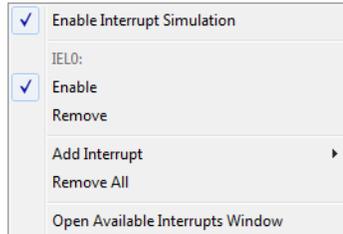
A timing variation range, as a percentage of the repeat interval, in which the interrupt might occur for a period. For example, if the repeat interval is 100 and the variance 5%, the interrupt might occur anywhere between $T=95$ and $T=105$, to simulate a variation in the timing. Click to edit.

Probability %

The probability, in percent, that the interrupt will actually occur within the specified period. Click to edit.

Context menu

This context menu is available:



These commands are available:

Enable Interrupt Simulation

Enables or disables the entire interrupt simulation system. 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.

Enable

Enables or disables the individual interrupt you clicked on.

Remove

Removes the individual interrupt you clicked on.

Add Interrupt

Selects an interrupt to install. The drop-down list contains all available interrupts. Your selection will automatically update the **Description** box. The list is populated with entries from the device description file that you have selected.

Remove All

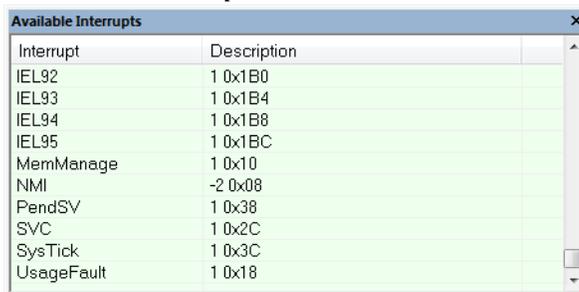
Removes all installed interrupts in the window.

Open Available Interrupts Window

Opens the **Available Interrupts** window, see *Available Interrupts window*, page 254.

Available Interrupts window

The **Available Interrupts** window is available from the C-SPY driver menu.



Interrupt	Description
IEL92	1 0x1B0
IEL93	1 0x1B4
IEL94	1 0x1B8
IEL95	1 0x1BC
MemManage	1 0x10
NMI	-2 0x08
PendSV	1 0x38
SVC	1 0x2C
SysTick	1 0x3C
UsageFault	1 0x18

Use this window for an overview of all available interrupts for your project. You can also use it for forcing an interrupt instantly. This is useful when you want to check your interrupt logic and interrupt routines. Just start typing an interrupt name and focus shifts to the first line found with that name.

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 sort the window contents, click on either the **Interrupt** or the **Description** column header. A second click on the same column header reverses the sort order.

To force an interrupt:

- 1 Enable the interrupt simulation system, see *Interrupt Configuration window*, page 251.
- 2 Activate the interrupt by choosing the **Force Interrupt** command from the context menu.

Requirements

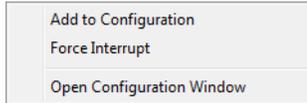
The C-SPY simulator.

Display area

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

Context menu

This context menu is available:



These commands are available:

Add to Configuration

Installs the selected interrupt and adds it to the **Interrupt Configuration** window.

Force Interrupt

Triggers the selected interrupt.

Open Configuration Window

Opens the **Interrupt Configuration** window, see *Interrupt Configuration window*, page 251.

Interrupt Status window

The **Interrupt Status** window is available from the C-SPY driver menu.

Interrupt	ID	Type	Status	Next Time	Timing [cycles]
TIM_INT	1	Single	Idle	0	0
NMI	0	Single	Idle	0	0
SCI0_I0	2	Repeat (macro)	Idle	4000	4000 + n*2000

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

Requirements

The C-SPY simulator.

Display area

This area contains these columns:

Interrupt

Lists all interrupts.

ID

A unique interrupt identifier.

Type

The type of the interrupt. The type can be one of:

Forced, a single-occasion interrupt defined in the **Available Interrupts** window.

Single, a single-occasion interrupt.

Repeat, a periodically occurring interrupt.

If the interrupt has been set from a C-SPY macro, the additional part (**macro**) is added, for example: **Repeat(macro)**.

Status

The state of the interrupt:

Idle, the interrupt activation signal is low (deactivated).

Pending, the interrupt activation signal is active, but the interrupt has not been yet acknowledged by the interrupt handler.

Executing, the interrupt is currently being serviced, that is the interrupt handler function is executing.

Suspended, the interrupt is currently suspended due to execution of an interrupt with a higher priority.

(deleted) is added to **Executing** and **Suspended** if you have deleted a currently active interrupt. **(deleted)** is removed when the interrupt has finished executing.

Next Time

The next time an idle interrupt is triggered. Once a repeatable interrupt starts 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: $\text{Activation Time} + n * \text{Repeat Time}$. 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 C-SPY driver menu.

Time	Interrupt	Status	Program Counter	Execution Time
109.32 us	IRQ0	Triggered	0x13E8	
111.26 us	IRQ0	Enter	0x13F0	
135.78 us	IRQ1	Enter	0x1126	
148.72 us	IRQ1	Leave	0x1378	12.94 us
189.34 us	Overflow			
207.30 us	IRQ0	Leave	0x1126	96.04 us
230.00 us	IRQ0	Triggered	0x1110	
231.34 us	IRQ0	Enter	0x1126	
240.26 us	IRQ0	Leave	0x1122	8.92 us
300.00 us	IRQ1	Enter	---	
371.12 us	IRQ1	Leave	0x1120	71.12 us
431.30 us	IRQ1	Enter	---	

Red indicates overflows and italic indicates approximate values

Light-colored rows indicate entrances to interrupts

Darker rows indicate exits from interrupts

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 oldest entries in the buffer are erased.

For more information, see *Getting started using interrupt logging*, page 250.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window—Interrupt Log graph*, page 263.

Requirements

The C-SPY simulator.

Display area

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 **Available Interrupts** 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

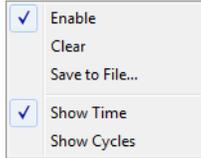
The value of the program counter when the event occurred.

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.

Context menu

This context menu is available:



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 also happen when you reset the debugger.

Save to File

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

Show Time

Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.

Show Cycles

Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Interrupt Log Summary window

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

Interrupt	Count	First Time	Total (Time)	Total (%)	Fastest	Slowest	Min Interval	Max Interval
ADC	5	25.560us	95.400us	17.61	16.320us	30.120us	192.640us	1284.100us
RTC	4	41.700us	55.200us	22.66	13.800us	13.800us	27.060us	2687.420us

Approximative time count: 1
 Overflow count: 1
 Current time: 3350.080us us

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

For more information, see *Getting started using interrupt logging*, page 250.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window—Interrupt Log graph*, page 263.

Requirements

The C-SPY simulator.

Display area

Each row in this area displays statistics about the specific interrupt based on the log information in these columns; and summary information is listed at the bottom of the display area:

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.

Total (%)

The time in percent of the current time.

Fastest**

The fastest execution of a single interrupt of this type.

Slowest**

The slowest execution of a single interrupt of this type.

Min Interval

The shortest time between two interrupts of this type.

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

Max Interval

The longest time between two interrupts of this type.

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

Approximative time count

The information displayed depends on the C-SPY driver you are using.

For some C-SPY drivers, this information is not displayed or the value is always zero. In this case, all logs have an exact time stamp.

For other C-SPY drivers, a non-zero value is displayed. The value represents the amount of logs with an approximative time stamp. This might happen if the bandwidth in the communication channel is too low compared to the amount of data packets generated by the CPU or if the CPU generated packets with an approximative time stamp.

Overflow count

The information displayed depends on the C-SPY driver you are using.

For some C-SPY drivers, this information is not displayed or the value is always zero.

For other C-SPY drivers, the number represents the amount of overflows in the communication channel which can cause logs to be lost. If this happens, it indicates that logs might be incomplete. To solve this, make sure not to use all C-SPY log features simultaneously or check used bandwidth for the communication channel.

Current time/Current cycles

The information displayed depends on the C-SPY driver you are using.

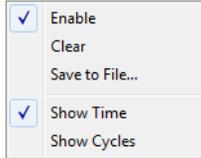
For some C-SPY drivers, the value is always zero or not visible at all.

For other C-SPY drivers, the number represents the current time or cycles—the number of cycles or the execution time since the start of execution.

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

Context menu

This context menu is available:



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 also happen when you reset the debugger.

Save to File

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

Show Time

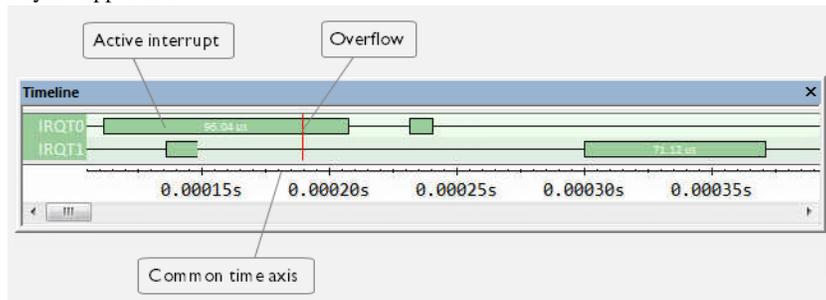
Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.

Show Cycles

Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Timeline window—Interrupt Log graph

The Interrupt Log graph displays interrupts collected by the trace system. In other words, the graph provides a graphical view of the interrupt events during the execution of your application.



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

Requirements

The C-SPY simulator.

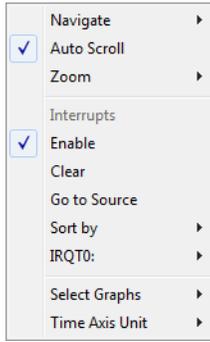
Display area

- The label area at the left end of the graph displays the names of the interrupts.
- The graph itself shows active interrupts as a thick green horizontal bar where the white figure indicates the time spent in the interrupt. This graph is a graphical representation of the information in the **Interrupt Log** window, see *Interrupt Log window*, page 257.
- If the bar is displayed without horizontal borders, there are two possible causes:
 - The interrupt is reentrant and has interrupted itself. Only the innermost interrupt will have borders.
 - There are irregularities in the interrupt enter-leave sequence, probably due to missing logs.
- If the bar is displayed without a vertical border, the missing border indicates an approximate time for the log.
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all interrupt logs from the target system.

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

Context menu

This context menu is available:



Note: The exact contents of the context menu you see on the screen depends on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Navigate

Commands for navigating 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

Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.

Zoom

Commands for zooming the window, in other words, changing the time scale. Choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +

Zoom Out zooms out on the time scale. Shortcut key: –

10ns, 100ns, 1us, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

1ms, 10ms, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window.

10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Interrupts

A heading that shows that the Interrupt Log-specific commands below are available.

Enable

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.

Clear

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

Go To Source

Displays the corresponding source code in an editor window, if applicable.

Sort by

Sorts the entries according to their ID or name. The selected order is used in the graph when new interrupts appear.

source

Goes to the previous/next log for the selected source.

Select Graphs

Selects which graphs to be displayed in the **Timeline** window.

Time Axis Unit

Selects the unit used in the time axis—choose between **Seconds** and **Cycles**.

If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.

C-SPY macros

- Introduction to C-SPY macros
- Using C-SPY macros
- Reference information on the macro language
- Reference information on reserved setup macro function names
- Reference information on C-SPY system macros
- Graphical environment for macros

Introduction to C-SPY macros

These topics are covered:

- Reasons for using C-SPY macros
- Briefly about using C-SPY macros
- Briefly about setup macro functions and files
- Briefly about the macro language

REASONS FOR USING C-SPY MACROS

You can use C-SPY macros either by themselves or in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. Some examples where macros can be useful:

- Automating the debug session, for instance with trace printouts, printing values of variables, and setting breakpoints.
- Hardware configuring, such as initializing hardware registers.
- Feeding your application with simulated data during runtime.
- Simulating peripheral devices, see the chapter *Interrupts*. This only applies if you are using the simulator driver.
- Developing small debug utility functions.

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 279.

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 274.

Example

Consider this example of a macro function which illustrates the various components of the macro language:

```
__var oldVal;
CheckLatest(val)
{
    if (oldVal != val)
    {
        __message "Message: Changed from ", oldVal, " to ", val, "\n";
        oldVal = val;
    }
}
```

Note: Reserved macro words begin with double underscores to prevent name conflicts.

Using C-SPY macros

These tasks are covered:

- Registering C-SPY macros—an overview
- Executing C-SPY macros—an overview
- Registering and executing using setup macros and setup files
- Executing macros using Quick Watch
- Executing a macro by connecting it to a breakpoint
- Aborting a C-SPY macro

For more examples using C-SPY macros, see:

- The tutorial about simulating an interrupt, which you can find in the Information Center
- *Initializing target hardware before C-SPY starts*, page 48

REGISTERING C-SPY MACROS—AN OVERVIEW

C-SPY must know that you intend to use your defined macro functions, and therefore you must *register* your macros. There are various ways to register macro functions:

- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files*, page 271.
- You can register macros interactively in the **Macro Registration** window, see *Macro Registration window*, page 327. Registered macros appear in the **Debugger Macros** window, see *Debugger Macros window*, page 329.
- 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 308.

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 271.
- 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 271.
- The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is more specifically designed for C-SPY macros. See *Macro Quicklaunch window*, page 331.
- 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 272.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to execute your macro.

REGISTERING AND EXECUTING USING SETUP MACROS AND SETUP FILES

It can be convenient to register a macro file during the C-SPY startup sequence. To do this, specify a macro file which you load before starting the debug session. Your macro functions will be automatically registered each time you start the debugger.

If you use the reserved setup macro function names to define the macro functions, you can define exactly at which stage you want the macro function to be executed.

To define a setup macro function and load it during C-SPY startup:

- 1 Create a new text file where you can define your macro function.

For example:

```
execUserSetup()
{
    ...
    __registerMacroFile("MyMacroUtils.mac");
    __registerMacroFile("MyDeviceSimulation.mac");
}
```

This macro function registers the additional macro files `MyMacroUtils.mac` and `MyDeviceSimulation.mac`. Because the macro function is defined with the function name `execUserSetup`, it will be executed directly after your application has been downloaded.

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

The macros will now be registered during the C-SPY startup sequence.

EXECUTING MACROS USING QUICK WATCH

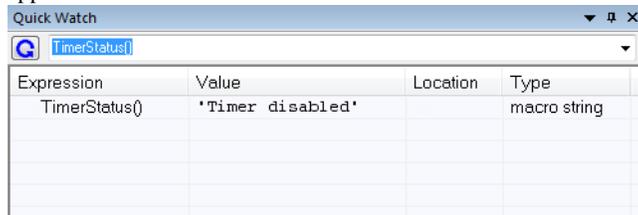
The **Quick Watch** window lets you dynamically choose when to execute a macro function.

- 1 Consider this simple macro function that checks the status of a timer enable bit:

```
TimerStatus()
{
    if ((TimerStatreg & 0x01) != 0) /* Checks the status of reg */
        return "Timer enabled"; /* C-SPY macro string used */
    else
        return "Timer disabled"; /* C-SPY macro string used */
}
```

- 2 Save the macro function using the filename extension `mac`.
- 3 To load the macro file, choose **View>Macros>Macro Registration**. The **Macro Registration** window is displayed. Click **Add** and locate the file using the file browser. The macro file appears in the list of macros in the **Macro Registration** window.
- 4 Select the macro you want to register and your macro will appear in the **Debugger Macros** window.
- 5 Choose **View>Quick Watch** to open the **Quick Watch** window, type the macro call `TimerStatus()` in the text field and press Return,

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



The macro will automatically be displayed in the **Quick Watch** window. For more information, see *Quick Watch window*, page 97.

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 **Debug Log** window.

Save the macro function in a macro file, with the filename extension `mac`.

- 3 To register the macro, choose **View>Macros>Macro Registration** to open the **Macro Registration** window and add your macro file to the list. Select the file to register it. Your macro function will appear in the **Debugger Macros** window.
- 4 To set a code breakpoint, click the **Toggle Breakpoint** button on the first statement within the function `fact` in your application source code. Choose **View>Breakpoints** to open the **Breakpoints** window. Select your breakpoint in the list of breakpoints and choose the **Edit** command from the context menu.
- 5 To connect the log macro function to the breakpoint, type the name of the macro function, `logfact ()`, in the **Action** field and click **OK** to 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 **Debug 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 120
 - Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 114.
- 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 277.

For an example where a serial port input buffer is simulated using the method of connecting a macro to a breakpoint, see the tutorial *Simulating an interrupt* in the Information Center.

ABORTING A C-SPY MACRO

To abort a C-SPY macro:

- 1 Press **Ctrl+Shift+.** (period) for a short while.
- 2 A message that says that the macro has terminated is displayed in the **Debug Log** window.

This method can be used if you suspect that something is wrong with the execution, for example because it seems not to terminate in a reasonable time.

Reference information on the macro language

Reference information about:

- *Macro functions*, page 274
- *Macro variables*, page 274
- *Macro parameters*, page 275
- *Macro strings*, page 275
- *Macro statements*, page 276
- *Formatted output*, page 277

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 78.

The syntax for defining one or more macro variables is:

```
__var nameList;
```

where *nameList* is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and it exists throughout the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and destroyed on return from the macro.

By default, macro variables are treated as signed integers and initialized to 0. When a C-SPY variable is assigned a value in an expression, it also acquires the type of that expression. For example:

Expression	What it means
<code>myvar = 3.5;</code>	<code>myvar</code> is now type <code>double</code> , value <code>3.5</code> .
<code>myvar = (int*)i;</code>	<code>myvar</code> is now type pointer to <code>int</code> , and the value is the same as <code>i</code> .

Table 12: Examples of C-SPY macro variables

In case of a name conflict between a C symbol and a C-SPY macro variable, C-SPY macro variables have a higher precedence than C variables. Note that macro variables are allocated on the debugger host and do not affect your application.

MACRO PARAMETERS

A macro parameter is intended for parameterization of device support. The named parameter will behave as a normal C-SPY macro variable with these differences:

- The parameter definition can have an initializer
- Values of a parameters can be set through options (either in the IDE or in `cspybat`).
- A value set from an option will take precedence over a value set by an initializer
- A parameter must have an initializer, be set through an option, or both. Otherwise, it has an undefined value, and accessing it will cause a runtime error.

The syntax for defining one or more macro parameters is:

```
__param param[=value, ...;]
```

Use the command line option `--macro_param` to specify a value to a parameter, see `--macro_param`, page 360.

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 277.

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 78.

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 301.

To produce messages in the **Debug Log** window:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2,
" in the Debug Log window.";
```

This produces this message in the **Debug Log** window:

This line prints the values 42 and 37 in the Debug 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.

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
```

Optionally, a number can be inserted between the % and the letter, to format an integer to that minimum width. Binary, octal, and hexadecimal numbers will be left-padded with zeros, decimal numbers and characters will be padded with spaces. Note that numbers that do not fit within the requested minimum width will *not* be truncated.

Examples:

```
__message 31:%4x;      // 001f
__message 31:%4d;      //  31
__message 31:%8b;      // 00011111
```

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.

Reference information about:

- `execUserAttach`
- `execUserPreload`
- `execUserExecutionStarted`
- `execUserExecutionStopped`
- `execUserFlashInit`
- `execUserSetup`
- `execUserFlashReset`
- `execUserPreReset`
- `execUserReset`
- `execUserExit`

- `execUserFlashExit`
- `execUserCoreConnect`

`execUserAttach`

Syntax	<code>execUserAttach</code>
For use with	The C-SPY I-jet driver.
Description	Called after the debugger attaches to a running application at its current location without resetting the target system (the option Attach to running target). Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.

`execUserPreload`

Syntax	<code>execUserPreload</code>
For use with	All C-SPY drivers.
Description	Called after communication with the target system is established but before downloading the target application. Implement this macro to initialize memory locations and/or registers which are vital for loading data properly. Note: Do not use this macro if you are using a flash loader. Use the macro <code>execUserFlashInit</code> instead to perform early initializations required by the flash loader, see <i>execUserFlashInit</i> , page 281.

`execUserExecutionStarted`

Syntax	<code>execUserExecutionStarted</code>
For use with	The C-SPY simulator.
Description	Called when the debugger is about to start or resume execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window.

execUserExecutionStopped

Syntax	<code>execUserExecutionStopped</code>
For use with	The C-SPY simulator.
Description	Called when the debugger has stopped execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window.

execUserFlashInit

Syntax	<code>execUserFlashInit</code>
For use with	The C-SPY hardware debugger drivers.
Description	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.

execUserSetup

Syntax	<code>execUserSetup</code>
For use with	All C-SPY drivers.
Description	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.



If you define interrupts or breakpoints in a macro file that is executed at system start (using `execUserSetup`) we strongly recommend that you also make sure that they are removed at system shutdown (using `execUserExit`). An example is available in `SetupSimple.mac`, see the tutorials in the Information Center.

The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time `execUserSetup` is executed again. This seriously affects the execution speed.

execUserFlashReset

Syntax	<code>execUserFlashReset</code>
For use with	The C-SPY hardware debugger drivers.
Description	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.

execUserPreReset

Syntax	<code>execUserPreReset</code>
For use with	All C-SPY drivers.
Description	Called each time just before the reset command is issued. Implement this macro to set up any required device state.

execUserReset

Syntax	<code>execUserReset</code>
For use with	All C-SPY drivers.
Description	Called each time just after the reset command is issued. Implement this macro to set up and restore data.

execUserExit

Syntax	<code>execUserExit</code>
For use with	All C-SPY drivers.
Description	Called once when the debug session ends. Implement this macro to save status data etc.

execUserFlashExit

Syntax	execUserFlashExit
For use with	The C-SPY hardware debugger drivers.
Description	Called once when the flash programming ends. Implement this macro to save status data etc. This macro is useful for flash loader functionality.

execUserCoreConnect

Syntax	execUserCoreConnect
For use with	The C-SPY I-jet driver.
Description	Called immediately when connection with the probe is established. Implement this macro to perform actions before connecting the CPU. This macro is useful for unlocking/erasing a secured device.

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>__argCount</code>	Returns the number of arguments in a string. See the documentation in the <i>Flash Loader Development Guide</i> .
<code>__abortLaunch</code>	Aborts the launch of the debugger
<code>__bytes2Word16</code>	Extracts a 16-bit word from a buffer. See the documentation in the <i>Flash Loader Development Guide</i> .
<code>__bytes2Word32</code>	Extracts a 32-bit word from a buffer. See the documentation in the <i>Flash Loader Development Guide</i> .
<code>__cancelAllInterrupts</code>	Cancels all ordered interrupts
<code>__cancelInterrupt</code>	Cancels an interrupt
<code>__clearBreak</code>	Clears a breakpoint
<code>__closeFile</code>	Closes a file that was opened by <code>__openFile</code>

Table 13: Summary of system macros

Macro	Description
<code>__delay</code>	Delays execution
<code>__disableInterrupts</code>	Disables generation of interrupts
<code>__driverType</code>	Verifies the driver type
<code>__enableInterrupts</code>	Enables generation of interrupts
<code>__evaluate</code>	Interprets the input string as an expression and evaluates it
<code>__fillMemory8</code>	Fills a specified memory area with a byte value
<code>__fillMemory16</code>	Fills a specified memory area with a 2-byte value
<code>__fillMemory32</code>	Fills a specified memory area with a 4-byte value
<code>__fillMemory64</code>	Fills a specified memory area with an 8-byte value
<code>__gdbserver_exec_command</code>	Send strings or commands to the GDB Server
<code>__getArg</code>	Returns an argument from a string. See the documentation in the <i>Flash Loader Development Guide</i> .
<code>__getNumberOfCores</code>	Gets the number local cores being debugged.
<code>__getSelectedCore</code>	Gets the number of the current core.
<code>__isBatchMode</code>	Checks if C-SPY is running in batch mode or not.
<code>__isMacroSymbolDefined</code>	Checks if a C-SPY macro symbol is defined.
<code>__loadImage</code>	Loads a debug image
<code>__makeString</code>	Creates a new buffer string. See the documentation in the <i>Flash Loader Development Guide</i> .
<code>__memoryRestore</code>	Restores the contents of a file to a specified memory zone
<code>__memorySave</code>	Saves the contents of a specified memory area to a file
<code>__messageBoxYesCancel</code>	Displays a Yes/Cancel dialog box for user interaction
<code>__messageBoxYesNo</code>	Displays a Yes/No dialog box for user interaction
<code>__openFile</code>	Opens a file for I/O operations
<code>__orderInterrupt</code>	Generates an interrupt
<code>__popSimulatorInterruptExecutingStack</code>	Informs the interrupt simulation system that an interrupt handler has finished executing
<code>__probeType</code>	Verifies the probe type
<code>__readFile</code>	Reads from the specified file
<code>__readFileByte</code>	Reads one byte from the specified file

Table 13: Summary of system macros (Continued)

Macro	Description
<code>__readMemory8,</code> <code>__readMemoryByte</code>	Reads one byte from the specified memory location
<code>__readMemory16</code>	Reads two bytes from the specified memory location
<code>__readMemory32</code>	Reads four bytes from the specified memory location
<code>__readMemory64</code>	Reads eight bytes from the specified memory location
<code>__readMemoryBuffer</code>	Reads bytes and returns them as a string. See the documentation in the <i>Flash Loader Development Guide</i> .
<code>__registerMacroFile</code>	Registers macros from the specified file
<code>__resetFile</code>	Rewinds a file opened by <code>__openFile</code>
<code>__selectCore</code>	Switches focus from the current core to the specified core.
<code>__setCodeBreak</code>	Sets a code breakpoint
<code>__setDataBreak</code>	Sets a data breakpoint
<code>__setDataLogBreak</code>	Sets a data log breakpoint
<code>__setLogBreak</code>	Sets a log breakpoint
<code>__setSimBreak</code>	Sets a simulation breakpoint
<code>__setTraceStartBreak</code>	Sets a trace start trigger breakpoint
<code>__setTraceStopBreak</code>	Sets a trace stop trigger 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>__system1</code>	Starts an external application
<code>__system2</code>	Starts an external application with <code>stdout</code> and <code>stderr</code> collected in one variable
<code>__system3</code>	Starts an external application with <code>stdout</code> and <code>stderr</code> collected in separate variables
<code>__targetDebuggerVersion</code>	Returns the version of the target debugger
<code>__toLower</code>	Returns a copy of the parameter string where all the characters have been converted to lower case
<code>__toString</code>	Prints strings

Table 13: Summary of system macros (Continued)

Macro	Description
<code>__toUpper</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>__wallTime_ms</code>	Returns the current host computer CPU time in milliseconds
<code>__writeFile</code>	Writes to the specified file
<code>__writeFileByte</code>	Writes one byte to the specified file
<code>__writeMemory8,</code> <code>__writeMemoryByte</code>	Writes one byte to the specified memory location
<code>__writeMemory16</code>	Writes a two-byte word to the specified memory location
<code>__writeMemory32</code>	Writes a four-byte word to the specified memory location
<code>__writeMemory64</code>	Writes an eight-byte word to the specified memory location
<code>__writeMemoryBuffer</code>	Writes bytes from a buffer. See the documentation in the <i>Flash Loader Development Guide</i> .

Table 13: Summary of system macros (Continued)

`__abortLaunch`

Syntax	<code>__abortLaunch (message)</code>
Parameters	<i>message</i> A string that is printed as an error message when the macro executes.
Return value	None.
For use with	All C-SPY drivers.
Description	This macro can be used for aborting a debugger launch, for example if another macro sees that something goes wrong during initialization and cannot perform a proper setup. This is an emergency stop when launching, not a way to end an ongoing debug session like the C library function <code>abort()</code> .

Example

```
if (!__messageBoxYesCancel("Do you want to mass erase to unlock
                           the device?", "Unlocking device"))
{ __abortLaunch("Unlock canceled. Debug session cannot
                continue."); }
```

__cancelAllInterrupts

Syntax	<code>__cancelAllInterrupts()</code>
Return value	<code>int 0</code>
For use with	The C-SPY Simulator.
Description	Cancels all ordered interrupts.

__cancelInterrupt

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

Return value

Result	Value
Successful	<code>int 0</code>
Unsuccessful	Non-zero error number

Table 14: `__cancelInterrupt` return values

For use with	The C-SPY Simulator.
Description	Cancels the specified interrupt.

__clearBreak

Syntax	<code>__clearBreak(<i>break_id</i>)</code>
Parameters	<i>break_id</i> The value returned by any of the set breakpoint macros.

Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Clears a user-defined breakpoint.
See also	<i>Breakpoints</i> , page 105.

__closeFile

Syntax	<code>__closeFile(<i>fileHandle</i>)</code>
Parameters	<i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Closes a file previously opened by <code>__openFile</code> .

__delay

Syntax	<code>__delay(<i>value</i>)</code>
Parameters	<i>value</i> The number of milliseconds to delay execution.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Delays execution the specified number of milliseconds.

__disableInterrupts

Syntax `__disableInterrupts()`

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 15: `__disableInterrupts` return values

For use with The C-SPY Simulator.

Description Disables the generation of interrupts.

__driverType

Syntax `__driverType(driver_id)`

Parameters

driver_id

A string corresponding to the driver you want to check for. Choose one of these:

"sim" corresponds to the simulator driver

"gdbserv" corresponds to the C-SPY GDB Server driver

"ijet" corresponds to the C-SPY I-jet driver

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 16: `__driverType` return values

For use with All C-SPY drivers.

Description Checks to see if the current C-SPY driver is identical to the driver type of the *driver_id* parameter.

Example

```
__driverType("sim")
```

If the simulator is the current driver, the value 1 is returned. Otherwise 0 is returned.

__enableInterrupts

Syntax `__enableInterrupts()`

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 17: `__enableInterrupts` return values

For use with The C-SPY Simulator.

Description Enables the generation of interrupts.

__evaluate

Syntax `__evaluate(string, valuePtr)`

Parameters

string

Expression string.

valuePtr

Pointer to a macro variable storing the result.

Return value

Result	Value
Successful	int 0
Unsuccessful	int 1

Table 18: `__evaluate` return values

For use with All C-SPY drivers.

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

__fillMemory8

Syntax	<code>__fillMemory8(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>								
Parameters	<p><i>value</i> An integer that specifies the value.</p> <p><i>address</i> An integer that specifies the memory start address.</p> <p><i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 130.</p> <p><i>length</i> An integer that specifies how many bytes are affected.</p> <p><i>format</i> A string that specifies the exact fill operation to perform. Choose between:</p> <table> <tr> <td>Copy</td> <td><i>value</i> will be copied to the specified memory area.</td> </tr> <tr> <td>AND</td> <td>An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> <tr> <td>OR</td> <td>An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> <tr> <td>XOR</td> <td>An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> </table>	Copy	<i>value</i> will be copied to the specified memory area.	AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.	OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.	XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
Copy	<i>value</i> will be copied to the specified memory area.								
AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
Return value	int 0								
For use with	All C-SPY drivers.								
Description	Fills a specified memory area with a byte value.								
Example	<code>__fillMemory8(0x80, 0x700, "Memory", 0x10, "OR");</code>								

__fillMemory16

Syntax	<code>__fillMemory16(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>
Parameters	<p><i>value</i> An integer that specifies the value.</p>

	<i>address</i>	An integer that specifies the memory start address.
	<i>zone</i>	A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 130.
	<i>length</i>	An integer that defines how many 2-byte entities to be affected.
	<i>format</i>	A string that specifies the exact fill operation to perform. Choose between:
	Copy	<i>value</i> will be copied to the specified memory area.
	AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
	OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
	XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
Return value	int 0	
For use with	All C-SPY drivers.	
Description	Fills a specified memory area with a 2-byte value.	
Example	<code>__fillMemory16(0xCDCD, 0x7000, "Memory", 0x200, "Copy");</code>	

__fillMemory32

Syntax	<code>__fillMemory32(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>	
Parameters	<i>value</i>	An integer that specifies the value.
	<i>address</i>	An integer that specifies the memory start address.
	<i>zone</i>	A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 130.

	<i>length</i>	An integer that defines how many 4-byte entities to be affected.
	<i>format</i>	A string that specifies the exact fill operation to perform. Choose between:
	Copy	<i>value</i> will be copied to the specified memory area.
	AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
	OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
	XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
Return value	int 0	
For use with	All C-SPY drivers.	
Description	Fills a specified memory area with a 4-byte value.	
Example	<code>__fillMemory32(0x0000FFFF, 0x4000, "Memory", 0x1000, "XOR");</code>	

__fillMemory64

Syntax	<code>__fillMemory64(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>	
Parameters	<i>value</i>	An integer that specifies the value.
	<i>address</i>	An integer that specifies the memory start address.
	<i>zone</i>	A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 130.
	<i>length</i>	An integer that defines how many 8-byte entities to be affected.

format

A string that specifies the exact fill operation to perform. Choose between:

Copy	<i>value</i> will be copied to the specified memory area.
AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.

Return value	int 0
For use with	All C-SPY drivers.
Description	Fills a specified memory area with an 8-byte value.
Example	<pre>__fillMemory64(0x0155'FFFF'FFFF'FFFF, 0x4000, "Memory", 0x1000, "AND");</pre>

__gdbserver_exec_command

Syntax	<code>__gdbserver_exec_command("string")</code>
Parameters	<i>string</i> String or command sent to the GDB Server. For more information, see the GDB server documentation.
For use with	The C-SPY GDB Server driver.
Description	Use this option to send strings or commands to the GDB Server.

__getNumberOfCores

Syntax	<code>__getNumberOfCores()</code>
Return value	The number of local cores being debugged.
For use with	The C-SPY simulator. The C-SPY I-jet driver.
Description	This macro returns the number of local cores being debugged.
Example	<pre>test () { __var i; for (i = 0; i < __getNumberOfCores(); i++) { __selectCore(i); __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n"; } }</pre>
See also	<code>__getSelectedCore</code> , page 295 and <code>__selectCore</code> , page 308

__getSelectedCore

Syntax	<code>__getSelectedCore()</code>
Return value	The current core. The cores are numbered from 0 and upwards.
For use with	The C-SPY simulator. The C-SPY I-jet driver.
Description	Gets the number of the current core.

Example

```
test ()
{
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x,
  "\n";
  __selectCore(0);
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x,
  "\n";
  __selectCore(1);
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x,
  "\n";
}
```

A typical result of the above macro would be (assuming that the original core was number 1):

```
Core: 1 pc = 0000213C
Core: 0 pc = 00000494
Core: 1 pc = 0000213C
```

See also

`__selectCore`, page 308.

__isBatchMode**Syntax**

```
__isBatchMode()
```

Return value

Result	Value
True	int 1
False	int 0

Table 19: `__isBatchMode` return values

For use with

All C-SPY drivers.

Description

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

__isMacroSymbolDefined**Syntax**

```
__isMacroSymbolDefined(symbol)
```

Parameters

symbol

The name of a C-SPY macro variable or macro function (a string).

Return value

1 if *symbol* is an existing macro symbol. 0 if *symbol* is not defined.

For use with	All C-SPY drivers.
Description	This macro identifies whether a string is the name of an existing C-SPY macro symbol (variable or function) or not.

Example	<pre> __var someVariable; ... if (__isMacroSymbolDefined("someVariable")) someVariable = 42; else __message "The someVariable symbol is not defined!"; </pre>
---------	---

__loadImage

Syntax	<code>__loadImage(<i>path</i>, <i>offset</i>, <i>debugInfoOnly</i>)</code>
Parameters	<p><i>path</i></p> <p>A string that identifies the path to the debug 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 RISC-V</i>.</p> <p><i>offset</i></p> <p>An integer that identifies the offset to the destination address for the downloaded debug image.</p> <p><i>debugInfoOnly</i></p> <p>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.</p>

Return value	<table border="1"> <thead> <tr> <th>Value</th> <th>Result</th> </tr> </thead> <tbody> <tr> <td>Non-zero integer number</td> <td>A unique module identification.</td> </tr> <tr> <td>int 0</td> <td>Loading failed.</td> </tr> </tbody> </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 20: __loadImage return values

For use with	All C-SPY drivers.
Description	<p>Loads a debug image (debug file).</p> <p>Note: Images are only downloaded to RAM and no flash loading will be performed.</p>

Example 1

Your system consists of a ROM library and an application. The application is your active project, but you have a debug file corresponding to the library. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage("ROMfile", 0x8000, 1);
```

This macro call loads the debug information for the ROM library `ROMfile` without downloading its contents (because it is presumably already in ROM). Then you can debug your application together with the library.

Example 2

Your system consists of a ROM library and an application, but your main concern is the library. The library needs to be programmed into flash memory before a debug session. While you are developing the library, the library project must be the active project in the IDE. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage("ApplicationFile", 0x8000, 0);
```

The macro call loads the debug information for the application and downloads its contents (presumably into RAM). Then you can debug your library together with the application.

See also

Images, page 376 and *Loading multiple debug images*, page 45.

__memoryRestore

Syntax

```
__memoryRestore(zone, filename, offset)
```

Parameters

zone

A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

filename

A string that specifies the file to be read. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide for RISC-V*.

offset

An integer offset. When restoring data from the file into memory, this offset is added to the addresses specified in the file. For example, if the file contains data from `0x0-0x1FF` and the offset is `0x400`, the data will be placed in memory in the range `0x400-0x5FF`. This makes it possible to restore data into memory on addresses larger than 32-bit, even if the file format only supports 32-bit addresses.

Return value	int 0
For use with	All C-SPY drivers.
Description	Reads the contents of a file and saves it to the specified memory zone.
Example	<code>__memoryRestore("Memory", "c:\\temp\\saved_mem.hex", 0x400);</code>
See also	<i>Memory Restore dialog box</i> , page 143.

__memorySave

Syntax	<code>__memorySave(start, stop, format, filename, zerostart)</code>
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> <ul style="list-style-type: none"> <code>intel-extended</code> <code>motorola</code> <code>motorola-s19</code> <code>motorola-s28</code> <code>motorola-s37</code> <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 RISC-V</i>.</p>

zerostart

An integer. If it is 1 (or any non-zero value), the addresses in the saved file will start from 0x0. For example, if the specified memory range is 0x400–0x5FF, the address range in the file will be 0x0–0x1FF. This makes it possible to save memory from addresses larger than 32-bit to file formats which only support 32-bit addresses. If the parameter is 0, the file will contain the specified addresses as given.

Return value	int 0
For use with	All C-SPY drivers.
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", 0);</pre>
See also	<i>Memory Save dialog box</i> , page 142.

__messageBoxYesCancel

Syntax	<code>__messageBoxYesCancel(<i>message</i>, <i>caption</i>)</code>
Parameters	<p><i>message</i></p> <p>A message that will appear in the message box.</p> <p><i>caption</i></p> <p>The title that will appear in the message box.</p>

Return value

Result	Value
Yes	1
No	0

Table 21: __messageBoxYesCancel return values

For use with	All C-SPY drivers.
Description	Displays a Yes/Cancel dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

__messageBoxYesNo

Syntax `__messageBoxYesNo(message, caption)`

Parameters *message*

A message that will appear in the message box.

caption

The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 22: __messageBoxYesNo return values

For use with All C-SPY drivers.

Description Displays a Yes/No dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

__openFile

Syntax `__openFile(filename, access)`

Parameters *filename*

The file to be opened. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide for RISC-V*.

access

The access type (string).

These are mandatory but mutually exclusive:

"a" append, new data will be appended at the end of the open file

"r" read (by default in text mode; combine with b for binary mode: rb)

"w" write (by default in text mode; combine with b for binary mode: wb)

These are optional and mutually exclusive:

"b" binary, opens the file in binary mode

"t" ASCII text, opens the file in text mode

This access type is optional:

"+" together with r, w, or a; r+ or w+ is *read* and *write*, while a+ is *read* and *append*

Return value

Result	Value
Successful	The file handle
Unsuccessful	An invalid file handle, which tests as False

Table 23: `__openFile` return values

For use with

All C-SPY drivers.

Description

Opens a file for I/O operations. The default base directory of this macro is where the currently open project file (*.ewp) is located. The argument to `__openFile` can specify a location relative to this directory. In addition, you can use argument variables such as `$PROJ_DIR$` and `$TOOLKIT_DIR$` in the path argument.

Example

```
__var myFileHandle;          /* The macro variable to contain */
                             /* the file handle */
myFileHandle = __openFile("$PROJ_DIR$\Debug\Exe\test.tst",
"r");
if (myFileHandle)
{
    /* successful opening */
}
```

See also

For information about argument variables, see the *IDE Project Management and Building Guide for RISC-V*.

`__orderInterrupt`

Syntax

```
__orderInterrupt(specification, first_activation,
                 repeat_interval, variance, infinite_hold_time,
                 hold_time, probability)
```

Parameters

specification

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 <i>specification</i> is incorrect, it returns -1.
For use with		The C-SPY simulator.
Description		Generates an interrupt.
Example		This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles: <pre>__orderInterrupt("USARTR_VECTOR", 4000, 2000, 0, 1, 0, 100);</pre>

__popSimulatorInterruptExecutingStack

Syntax	<code>__popSimulatorInterruptExecutingStack(void)</code>
Return value	<code>int 0</code>
For use with	The C-SPY simulator.
Description	<p>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.</p> <p>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</p>

task-switching. In this case, the interrupt simulation system cannot automatically detect that the interrupt has finished executing.

See also

Simulating an interrupt in a multi-task system, page 249.

__probeType

Syntax

`__probeType(probe_id)`

Parameters

probe_id

A string corresponding to the probe you want to check for. Choose one of these:

"I-jet" corresponds to the I-jet probe

"I-jet-Trace" corresponds to the I-jet Trace probe.

The strings are case-insensitive.

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 24: `__probeType` return values

For use with

The C-SPY I-jet driver.

Description

Checks to see if the current connected probe is identical to the probe type of the *probe_id* parameter.

Example

`__probeType("I-jet")`

If I-jet is the current connected probe, the value 1 is returned. Otherwise 0 is returned.

__readFile

Syntax

`__readFile(fileHandle, valuePtr)`

Parameters

fileHandle

A macro variable used as filehandle by the `__openFile` macro.

valuePtr

A pointer to a variable.

Return value

Result	Value
Successful	0
Unsuccessful	Non-zero error number

Table 25: `__readFile` return values

For use with

All C-SPY drivers.

Description

Reads a sequence of hexadecimal digits from the given file and converts them to an `unsigned long` which is assigned to the `value` parameter, which should be a pointer to a macro variable.

Only printable characters representing hexadecimal digits and white-space characters are accepted, no other characters are allowed.

Example

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

In this example, if the file pointed to by `myFileHandle` contains the ASCII characters `1234 abcd 90ef`, consecutive reads will assign the values `0x1234 0xabcd 0x90ef` to the variable `number`.

`__readFileByte`

Syntax

```
__readFileByte(fileHandle)
```

Parameters

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.

For use with

All C-SPY drivers.

Description

Reads one byte from a file.

Example

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

__readMemory8, __readMemoryByte

Syntax	<code>__readMemory8(address, zone)</code> <code>__readMemoryByte(address, zone)</code>
Parameters	<i>address</i> The memory address (integer). <i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 130.
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads one byte from a given memory location.
Example	<code>__readMemory8(0x0108, "Memory");</code>

__readMemory16

Syntax	<code>__readMemory16(address, zone)</code>
Parameters	<i>address</i> The memory address (integer). <i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 130.
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads a two-byte word from a given memory location.
Example	<code>__readMemory16(0x0108, "Memory");</code>

__readMemory32

Syntax	<code>__readMemory32(address, zone)</code>
Parameters	<p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 130.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads a four-byte word from a given memory location.
Example	<code>__readMemory32(0x0108, "Memory");</code>

__readMemory64

Syntax	<code>__readMemory64(address, zone)</code>
Parameters	<p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 130.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads an eight-byte word from a given memory location.
Example	<code>__readMemory64(0x8000, "Memory");</code>

__registerMacroFile

Syntax	<code>__registerMacroFile(<i>filename</i>)</code>
Parameters	<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 RISC-V</i> .
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.
Example	<code>__registerMacroFile("c:\\testdir\\macro.mac");</code>
See also	<i>Using C-SPY macros</i> , page 269.

__resetFile

Syntax	<code>__resetFile(<i>fileHandle</i>)</code>
Parameters	<i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Rewinds a file previously opened by <code>__openFile</code> .

__selectCore

Syntax	<code>__selectCore(int <i>core</i>)</code>
Parameters	<i>core</i> The core to switch to. The cores are numbered from 0 and upwards.

Return value	int 0
For use with	The C-SPY simulator. The C-SPY I-jet driver.
Description	Switches focus from the current core to the specified core for the duration of the macro invocation or until any next invocation of <code>__selectCore</code> .
Example	<pre>test () { __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n"; __selectCore(0); __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n"; __selectCore(1); __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n"; }</pre> <p>A typical result of the above macro would be (assuming that the original core was number 1):</p> <pre>Core: 1 pc = 0000213C Core: 0 pc = 00000494 Core: 1 pc = 0000213C</pre>
See also	<code>__getSelectedCore</code> , page 295.

`__setCodeBreak`

Syntax	<code>__setCodeBreak(<i>location</i>, <i>count</i>, <i>condition</i>, <i>cond_type</i>, <i>action</i>)</code>
Parameters	<p><i>location</i></p> <p>A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see <i>Enter Location dialog box</i>, page 126.</p> <p><i>count</i></p> <p>An integer that specifies the number of times that a breakpoint condition must be fulfilled before a break occurs the next time.</p>

condition

The breakpoint condition. This must be a valid C-SPY expression, for instance a C-SPY macro function.

cond_type

The condition type; either "CHANGED" or "TRUE" (string).

action

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 26: `__setCodeBreak` return values

For use with

All C-SPY drivers.

Description

Sets a code breakpoint, that is, a breakpoint which is triggered just before the processor fetches an instruction at the specified location.

Examples

```
__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", "");
```

This example sets the breakpoint within a specific source file and line without using the absolute file path to the source:

```
__setCodeBreak("{main.c}.288.7", 0, "1", "TRUE", "");
```

See also

Breakpoints, page 105.

__setDataBreak

Syntax

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

Parameters

location

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For information about the location types, see *Enter Location dialog box*, page 126.

count

An integer that specifies the number of times that a breakpoint condition must be fulfilled before a break occurs the next time.

condition

The breakpoint condition (string).

cond_type

The condition type; either "CHANGED" or "TRUE" (string).

access

The memory access type: "R", for read, "W" for write, or "RW" for read/write.

action

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 27: *__setDataBreak* return values

For use with

The C-SPY simulator.

Description

Sets a data breakpoint, that is, a breakpoint which is triggered directly after the processor has read or written data at the specified location.

Example

```
__var brk;
brk = __setDataBreak("Memory:0x4710", 3, "d>6", "TRUE",
    "W", "ActionData()");
...
__clearBreak(brk);
```

See also

Breakpoints, page 105.

__setDataLogBreak

Syntax	<code>__setDataLogBreak(variable, access)</code>						
Parameters	<p><i>variable</i></p> <p>A string that defines the variable the breakpoint is set on, a variable of integer type with static storage duration. The microcontroller must also be able to access the variable with a single-instruction memory access, which means that you can only set data log breakpoints on 8-bit variables.</p> <p><i>access</i></p> <p>The memory access type: "R", for read, "W" for write, or "RW" for read/write.</p>						
Return value	<table border="1"> <thead> <tr> <th>Result</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Successful</td> <td>An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.</td> </tr> <tr> <td>Unsuccessful</td> <td>0</td> </tr> </tbody> </table> <p><i>Table 28: __setDataLogBreak return values</i></p>	Result	Value	Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.	Unsuccessful	0
Result	Value						
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.						
Unsuccessful	0						
For use with	The C-SPY simulator.						
Description	Sets a data log breakpoint, that is, a breakpoint which is triggered when a specified variable is accessed. Note that a data log breakpoint does not stop the execution, it just generates a data log.						
Example	<pre>__var brk; brk = __setDataLogBreak("MyVar", "R"); ... __clearBreak(brk);</pre>						
See also	<i>Breakpoints</i> , page 105 and <i>Getting started using data logging</i> , page 201.						

__setLogBreak

Syntax

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

Parameters

location

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box*, page 126.

message

The message text.

msg_type

The message type; choose between:

TEXT, the message is written word for word.

ARGS, the message is interpreted as a comma-separated list of C-SPY expressions or strings.

condition

The breakpoint condition (string).

cond_type

The condition type; either "CHANGED" or "TRUE" (string).

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	0

Table 29: __setLogBreak return values

For use with

All C-SPY drivers.

Description

Sets a log breakpoint, that is, a breakpoint which is triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily halt and print the specified message in the C-SPY **Debug Log** window.

Example

```

__var logBp1;
__var logBp2;

logOn()
{
    logBp1 = __setLogBreak ("{C:\\temp\\Utilities.c}.23.1",
        "\"Entering trace zone at :\", #PC:%X", "ARGS", "1", "TRUE");
    logBp2 = __setLogBreak ("{C:\\temp\\Utilities.c}.30.1",
        "Leaving trace zone...", "TEXT", "1", "TRUE");
}

logOff()
{
    __clearBreak(logBp1);
    __clearBreak(logBp2);
}

```

See also

Formatted output, page 277 and *Breakpoints*, page 105.

__setSimBreak**Syntax**

```
__setSimBreak(location, access, action)
```

Parameters

location

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For information about the location types, see *Enter Location dialog box*, page 126.

access

The memory access type: "R" for read or "W" for write.

action

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 30: __setSimBreak return values

For use with

The C-SPY simulator.

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

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

__setTraceStartBreak

Syntax `__setTraceStartBreak(location)`

Parameters *location*

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box*, page 126.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	0

Table 31: __setTraceStartBreak return values

For use with The C-SPY simulator.

The C-SPY I-jet driver and a device that supports trace.

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

Example

```

__var startTraceBp;
__var stopTraceBp;

traceOn()
{
    startTraceBp = __setTraceStartBreak
        ("C:\\TEMP\\Utilities.c).23.1");
    stopTraceBp = __setTraceStopBreak
        ("C:\\temp\\Utilities.c).30.1");
}

traceOff()
{
    __clearBreak(startTraceBp);
    __clearBreak(stopTraceBp);
}

```

See also

Trace Start Trigger breakpoint dialog box, page 191.

__setTraceStopBreak

Syntax

```
__setTraceStopBreak(location)
```

Parameters

location

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box*, page 126.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. The same value must be used when you want to clear the breakpoint.
Unsuccessful	int 0

Table 32: __setTraceStopBreak return values

For use with

The C-SPY simulator.
The C-SPY I-jet driver and a device that supports trace.

Description

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

Example

See *__setTraceStartBreak*, page 315.

See also *Trace Stop Trigger breakpoint dialog box*, page 193.

__sourcePosition

Syntax `__sourcePosition(linePtr, colPtr)`

Parameters

linePtr
Pointer to the variable storing the line number

colPtr
Pointer to the variable storing the column number

Return value

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

Table 33: *__sourcePosition* return values

For use with All C-SPY drivers.

Description If the current execution location corresponds to a source location, this macro returns the filename as a string. It also sets the value of the variables, pointed to by the parameters, to the line and column numbers of the source location.

__strFind

Syntax `__strFind(macroString, pattern, position)`

Parameters

macroString
A macro string.

pattern
The string pattern to search for

position
The position where to start the search. The first position is 0

Return value The position where the pattern was found or -1 if the string is not found.

For use with All C-SPY drivers.

Description	This macro searches a given string (<i>macroString</i>) for the occurrence of another string (<i>pattern</i>).
Example	<pre>__strFind("Compiler", "pile", 0) = 3 __strFind("Compiler", "foo", 0) = -1</pre>
See also	<i>Macro strings</i> , page 275.

__subString

Syntax	<code>__subString(<i>macroString</i>, <i>position</i>, <i>length</i>)</code>
Parameters	<p><i>macroString</i> A macro string.</p> <p><i>position</i> The start position of the substring. The first position is 0.</p> <p><i>length</i> The length of the substring</p>
Return value	A substring extracted from the given macro string.
For use with	All C-SPY drivers.
Description	This macro extracts a substring from another string (<i>macroString</i>).
Example	<pre>__subString("Compiler", 0, 2)</pre> <p>The resulting macro string contains Co.</p> <pre>__subString("Compiler", 3, 4)</pre> <p>The resulting macro string contains pile.</p>
See also	<i>Macro strings</i> , page 275.

__system1

Syntax	<code>__system1(<i>string</i>)</code>
Parameters	<p><i>string</i></p> <p>The command line used to start an external application. In some cases, the full path is needed. If it contains space characters, quotation marks escaped with backslashes (\ ") can be added to encapsulate the path, and, separately, the arguments to the application, like this:</p> <pre>"\"D:\My projects\my app\app.exe\" \"some argument\"".</pre>
Return value	The exit code returned from the external application. If the application could not be launched or fails to return an appropriate exit code, 1 is returned.
For use with	All C-SPY drivers.
Description	This macro launches an external application. It ignores all output returned from the application. Terminates the launched application if the application has not finished within 10 seconds.
Example	<pre>__var exitCode; exitCode = __system1("mkdir tmp");</pre>

__system2

Syntax	<code>__system2(<i>string</i>, &<i>output</i>)</code>
Parameters	<p><i>string</i></p> <p>The command line used to start an external application. In some cases, the full path is needed. If it contains space characters, quotation marks escaped with backslashes (\ ") can be added to encapsulate the path, and, separately, the arguments to the application, like this:</p> <pre>"\"D:\My projects\my app\app.exe\" \"some argument\"".</pre> <p><i>output</i></p> <p>The output returned from the application. Both the <code>stdout</code> and the <code>stderr</code> streams are stored in this variable.</p>
Return value	The exit code returned from the external application. If the application could not be launched or fails to return an appropriate exit code, 1 is returned.
For use with	All C-SPY drivers.

Description This macro launches an external application. The output from both the `stdout` and the `stderr` streams is stored in `output`. If no data has been received from the launched application within 10 seconds, or when the returned data exceeds 65535 bytes, the application is terminated. This restriction prevents the Embedded Workbench IDE from freezing or crashing because of misbehaving applications.

Example

```
__var exitCode;
__var out_err;

exitCode = __system2("dir /S", &out_err);

__message "Output from the dir command:";
__message out_err;
```

__system3

Syntax `__system3(string, &output, &error)`

Parameters

string

The command line used to start an external application. In some cases, the full path is needed. If it contains space characters, quotation marks escaped with backslashes (\") can be added to encapsulate the path, and, separately, the arguments to the application, like this:

```
"\"D:\My projects\my app\app.exe\" \"some argument\"".
```

output

The output returned from the `stdout` output stream of the application.

error

The output returned from the `stderr` output stream of the application.

Return value The exit code returned from the external application. If the application could not be launched or fails to return an appropriate exit code, 1 is returned.

For use with All C-SPY drivers.

Description This macro launches an external application. The output from the `stdout` stream is stored in `output` and the `stderr` stream is stored in `error`. If no data has been received from the launched application within 10 seconds, or when the returned data exceeds 65535 bytes, the application is terminated. This restriction prevents the Embedded Workbench IDE from freezing or crashing because of misbehaving applications.

Example

```

__var exitCode;
__var out;
__var err;

exitCode = __system3("dir /S", &out, &err);

__message "Output from the dir command:";
__message out;

__message "Error text from the dir command:";
__message err;

```

__targetDebuggerVersion

Syntax

```
__targetDebuggerVersion()
```

Return value

A string that represents the version number of the C-SPY debugger processor module.

For use with

All C-SPY drivers.

Description

This macro returns the version number of the C-SPY debugger processor module.

Example

```

__var toolVer;
toolVer = __targetDebuggerVersion();
__message "The target debugger version is, ", toolVer;

```

__toLower

Syntax

```
__toLower(macroString)
```

Parameters

macroString

A macro string.

Return value

The converted macro string.

For use with

All C-SPY drivers.

Description

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

Example

```
__toLower("IAR")
```

The resulting macro string contains `iar`.

```
__toLower("Mix42")
```

The resulting macro string contains `mix42`.

See also

Macro strings, page 275.

__toString

Syntax

```
__toString(C_string, maxlength)
```

Parameters

C_string

Any null-terminated C string.

maxlength

The maximum length of the returned macro string.

Return value

Macro string.

For use with

All C-SPY drivers.

Description

This macro is used for converting C strings (`char*` or `char[]`) into macro strings.

Example

Assuming your application contains this definition:

```
char const * hpstr = "Hello World!";
```

this macro call:

```
__toString(hpstr, 5)
```

would return the macro string containing `Hello`.

See also

Macro strings, page 275.

__ToUpper

Syntax

```
__ToUpper(macroString)
```

Parameters

macroString

A macro string.

Return value

The converted string.

For use with

All C-SPY drivers.

Description	This macro returns a copy of the parameter <i>macroString</i> where all the characters have been converted to upper case.
Example	<code>__toUpper("string")</code> The resulting macro string contains <code>STRING</code> .
See also	<i>Macro strings</i> , page 275.

__unloadImage

Syntax	<code>__unloadImage(module_id)</code>
Parameters	<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

Value	Result
<i>module_id</i>	A unique module identification (the same as the input parameter).
<code>int 0</code>	The unloading failed.

Table 34: `__unloadImage` return values

For use with	All C-SPY drivers.
Description	Unloads debug information from an already downloaded debug image.
See also	<i>Loading multiple debug images</i> , page 45 and <i>Images</i> , page 376.

__wallTime_ms

Syntax	<code>__wallTime_ms()</code>
Return value	Returns the current host computer CPU time in milliseconds.
For use with	All C-SPY drivers.
Description	This macro returns the current host computer CPU time in milliseconds. The first call will always return 0.

Example

```

__var t1;
__var t2;

t1 = __wallTime_ms();
__var i;
for (i =0; i < 1000; i++)
    __message "Tick";
t2 = __wallTime_ms();
__message "Elapsed time: ", t2 - t1;

```

__writeFile

Syntax

```
__writeFile(fileHandle, value)
```

Parameters

fileHandle

A macro variable used as filehandle by the `__openFile` macro.

value

An integer.

Return value

int 0

For use with

All C-SPY drivers.

Description

Prints the integer value in hexadecimal format (with a trailing space) to the file *file*.

Note: The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.

__writeFileByte

Syntax

```
__writeFileByte(fileHandle, value)
```

Parameters

fileHandle

A macro variable used as filehandle by the `__openFile` macro.

value

An integer.

Return value

int 0

For use with

All C-SPY drivers.

Description Writes one byte to the file *fileHandle*.

__writeMemory8, __writeMemoryByte

Syntax `__writeMemory8(value, address, zone)`
 `__writeMemoryByte(value, address, zone)`

Parameters

value
 An integer.

address
 The memory address (integer).

zone
 A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value `int 0`

For use with All C-SPY drivers.

Description Writes one byte to a given memory location.

Example `__writeMemory8(0x2F, 0x8020, "Memory");`

__writeMemory16

Syntax `__writeMemory16(value, address, zone)`

Parameters

value
 An integer.

address
 The memory address (integer).

zone
 A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value `int 0`

For use with All C-SPY drivers.

Description Writes two bytes to a given memory location.

Example `__writeMemory16(0x2FFF, 0x8020, "Memory");`

__writeMemory32

Syntax `__writeMemory32(value, address, zone)`

Parameters

value

An integer.

address

The memory address (integer).

zone

A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value `int 0`

For use with All C-SPY drivers.

Description Writes four bytes to a given memory location.

Example `__writeMemory32(0x5555FFFF, 0x8020, "Memory");`

__writeMemory64

Syntax `__writeMemory64(value, address, zone)`

Parameters

value

An integer.

address

The memory address (integer).

zone

A string that specifies the memory zone, see *C-SPY memory zones*, page 130.

Return value `int 0`

For use with All C-SPY drivers.

Description Writes eight bytes to a given memory location.

Example

```
__writeMemory64(0xFFFF'FFFF'8000'0000, 0xFFFF'8000, "Memory");
```

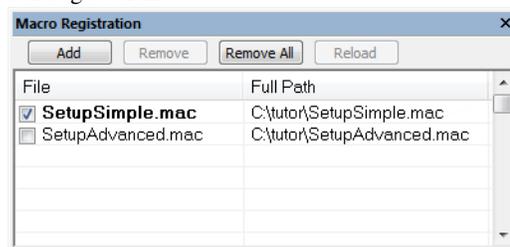
Graphical environment for macros

Reference information about:

- *Macro Registration window*, page 327
- *Debugger Macros window*, page 329
- *Macro Quicklaunch window*, page 331

Macro Registration window

The **Macro Registration** window is available from the **View>Macros** submenu during a debug session.



Use this window to list, register, and edit your debugger macro files.

Double-click a macro file to open it in the editor window and edit it.

See also *Registering C-SPY macros—an overview*, page 270.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

This area contains these columns:

File

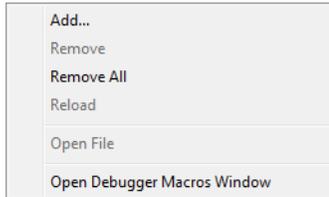
The name of an available macro file. To register the macro file, select the check box to the left of the filename. The name of a registered macro file appears in bold style.

Full path

The path to the location of the added macro file.

Context menu

This context menu is available:



These commands are available:

Add

Opens a file browser where you can locate the macro file that you want to add to the list. This menu command is also available as a function button at the top of the window.

Remove

Removes the selected debugger macro file from the list. This menu command is also available as a function button at the top of the window.

Remove All

Removes all macro files from the list. This menu command is also available as a function button at the top of the window.

Reload

Registers the selected macro file. Typically, this is useful when you have edited a macro file. This menu command is also available as a function button at the top of the window.

Open File

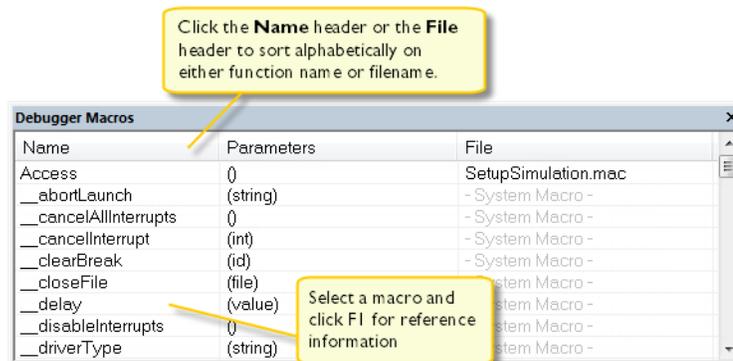
Opens the selected macro file in the editor window.

Open Debugger Macros Window

Opens the **Debugger Macros** window.

Debugger Macros window

The **Debugger Macros** window is available from the **View>Macros** submenu during a debug session.



Use this window to list all registered debugger macro functions, either predefined system macros or your own. This window is useful when you edit your own macro functions and want an overview of all available macros that you can use.

- Click the column headers **Name** or **File** to sort alphabetically on either function name or filename.
- Double-clicking a macro defined in a file opens that file in the editor window.
- To open a macro in the **Macro Quicklaunch** window, drag it from the **Debugger Macros** window and drop it in the **Macro Quicklaunch** window.
- Select a macro and press F1 to get online help information for that macro.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

This area contains these columns:

Name

The name of the debugger macro.

Parameters

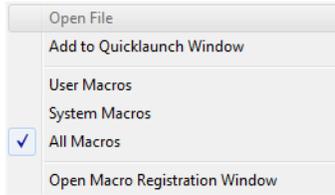
The parameters of the debugger macro.

File

For macros defined in a file, the name of the file is displayed. For predefined system macros, `-System Macro-` is displayed.

Context menu

This context menu is available:



These commands are available:

Open File

Opens the selected debugger macro file in the editor window.

Add to Quicklaunch Window

Adds the selected macro to the **Macro Quicklaunch** window.

User Macros

Lists only the debugger macros that you have defined yourself.

System Macros

Lists only the predefined system macros.

All Macros

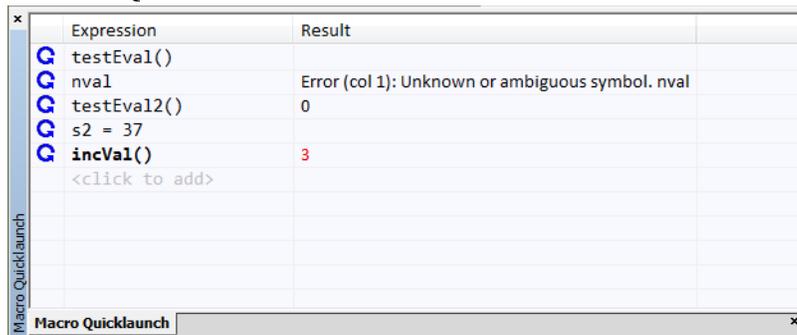
Lists all debugger macros, both predefined system macros and your own.

Open Macro Registration Window

Opens the **Macro Registration** window.

Macro Quicklaunch window

The **Macro Quicklaunch** window is available from the **View** menu.



Use this window to evaluate expressions, typically C-SPY macros.

For some devices, there are predefined C-SPY macros available with device support, typically provided by the chip manufacturer. These macros are useful for performing certain device-specific tasks. The macros are available in the **Macro Quicklaunch** window and are easily identified by their green icon.

The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is primarily designed for evaluating C-SPY macros. The window gives you precise control over when to evaluate an expression.

See also *Executing C-SPY macros—an overview*, page 270.

To add an expression:

- I Choose one of these alternatives:
 - Drag the expression to the window
 - In the **Expression** column, type the expression you want to examine.

If the expression you add and want to evaluate is a C-SPY macro, the macro must first be registered, see *Registering C-SPY macros—an overview*, page 270.

To evaluate an expression:

- I  Double-click the **Recalculate** icon to calculate the value of that expression.

Requirements

Can be used with all C-SPY debugger drivers and debug probes.

Display area

This area contains these columns:



Recalculate icon

To evaluate the expression, double-click the icon. The latest evaluated expression appears in bold style.

Expression

One or several expressions that you want to evaluate. Click <click to add> to add an expression. If the return value has changed since last time, the value will be displayed in red.

Result

Shows the return value from the expression evaluation.

Context menu

This context menu is available:



These commands are available:

Evaluate Now

Evaluates the selected expression.

Remove

Removes the selected expression.

Remove All

Removes all selected expressions.

The C-SPY command line utility—`cspybat`

- Using C-SPY in batch mode
- Summary of C-SPY command line options
- Reference information on C-SPY command line options

Using C-SPY in batch mode

You can execute C-SPY in batch mode if you use the command line utility `cspybat`, installed in the directory `common\bin`.

These topics are covered:

- Starting `cspybat`
- Output
- Invocation syntax

STARTING CSPYBAT

- 1 To start `cspybat` you must first create a batch file. An easy way to do that is to use one of the batch files that C-SPY automatically generates when you start C-SPY in the IDE.

C-SPY generates a batch file `projectname.buildconfiguration.cspy.bat` every time C-SPY is initialized. In addition, two more files are generated:

- `project.buildconfiguration.general.xml`, which contains options specific to `cspybat`
- `project.buildconfiguration.driver.xml`, which contains options specific to the C-SPY driver you are using

You can find the files in the directory `$PROJ_DIR$\settings`. The files contain the same settings as the IDE, and provide hints about additional options that you can use.

- 2 To start `cspybat`, you can use this command line:

```
project.cspybat.bat [debugfile]
```

Note that *debugfile* is optional. You can specify it if you want to use a different debug file than the one that is used in the *project.buildconfiguration.general.xcl* file.

OUTPUT

When you run `cspybat`, these types of output can be produced:

- *Terminal output from cspybat itself*
All such terminal output is directed to `stderr`. Note that if you run `cspybat` from the command line without any arguments, the `cspybat` version number and all available options including brief descriptions are directed to `stdout` and displayed on your screen.
- *Terminal output from the application you are debugging*
All such terminal output is directed to `stdout`, provided that you have used the `--plugin` option. See *--plugin*, page 361.
- *Error return codes*
`cspybat` returns status information to the host operating system that can be tested in a batch file. For *successful*, the value `int 0` is returned, and for *unsuccessful* the value `int 1` is returned.

INVOCATION SYNTAX

The invocation syntax for `cspybat` is:

```
cspybat processor_DLL driver_DLL debug_file
        [cspybat_options] --backend driver_options
```

Note: In those cases where a filename is required—including the DLL files—you are recommended to give a full path to the filename.

Parameters

The parameters are:

Parameter	Description
<i>processor_DLL</i>	The processor-specific DLL file—available in <code>riscv\bin</code> .
<i>driver_DLL</i>	The C-SPY driver DLL file—available in <code>riscv\bin</code> .
<i>debug_file</i>	The object file that you want to debug (filename extension out). See also <i>--debug_file</i> , page 342.

Table 35: *cspybat* parameters

Parameter	Description
<i>csybat_options</i>	The command line options that you want to pass to <code>csybat</code> . Note that these options are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 338.
<code>--backend</code>	Marks the beginning of the parameters to the C-SPY driver; all options that follow will be sent to the driver. Note that this option is mandatory.
<i>driver_options</i>	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 338.

Table 35: *csybat* parameters (Continued)

Summary of C-SPY command line options

Reference information about:

- General `csybat` options
- Options available for all C-SPY drivers
- Options available for the simulator driver
- Options available for all hardware debugger drivers
- Options available for the C-SPY I-jet driver
- Options available for the C-SPY GDB Server driver
- Options available for the C-SPY third-party drivers

GENERAL CSPYBAT OPTIONS

<code>--application_args</code>	Passes command line arguments to the debugged application.
<code>--attach_to_running_target</code>	Makes the debugger attach to a running application at its current location, without resetting the target system.
<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>--debug_file</code>	Specifies an alternative debug file.
<code>--device_macro</code>	Specifies a C-SPY device macro file.
<code>--download_only</code>	Downloads a code image without starting a debug session afterwards.
<code>-f</code>	Extends the command line.
<code>--flash_loader</code>	Specifies a flash loader specification XML file.
<code>--leave_target_running</code>	Makes the debugger leave the application running on the target hardware after the debug session is closed.
<code>--macro</code>	Specifies a macro file to be used.
<code>--macro_param</code>	Assigns a value to a C-SPY macro parameter.
<code>--plugin</code>	Specifies a plugin file to be used.
<code>--silent</code>	Omits the sign-on message.
<code>--suppress_entrypoint_warning</code>	Disables the warning when the ELF entry point is at address 0x0.
<code>--timeout</code>	Limits the maximum allowed execution time.

OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

<code>--core</code>	Specifies the core to be used.
<code>-p</code>	Specifies the device description file to be used.

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

<code>--disable_interrupts</code>	Disables the interrupt simulation.
<code>--disable_misalignment_exception</code>	Allows accesses to misaligned data.
<code>--function_profiling</code>	Analyzes your source code to find where the most time is spent during execution.
<code>--mapu</code>	Activates memory access checking.

--multicore_nr_of_cores Specify the number of cores on the device for multicore debugging.

OPTIONS AVAILABLE FOR ALL HARDWARE DEBUGGER DRIVERS

--drv_communication Specifies the communication link to be used.

--drv_communication_log Creates a log file.

--drv_default_breakpoint Sets the type of breakpoint resource to be used when setting breakpoints.

--drv_exclude_from_verify Excludes memory ranges from being verified.

--drv_reset_to_cpu_start Omits setting the PC when resetting the application.

--drv_restore_breakpoints Restores automatically any breakpoints that were destroyed during system startup.

--drv_vector_table_base Specifies the location of the Cortex-M reset vector and the initial stack pointer value.

OPTIONS AVAILABLE FOR THE C-SPY I-JET DRIVER

--drv_catch_exceptions Makes the application stop for certain exceptions.

--drv_interface Selects the communication interface.

--drv_interface_speed Specifies the JTAG interface speed.

--drv_suppress_download Suppresses download of the executable image. For reference information, see *Download*, page 375, specifically the option **Suppress download**.

--drv_system_bus_access Enables Live Watch and Live Memory reads for devices that support system bus access.

--drv_verify_download Verifies the target program. For reference information, see *Download*, page 375, specifically the option **Verify download**.

--jet_board_cfg Specifies a probe configuration file.

--jet_board_did Selects which CPU to debug on a multicore system.

<code>--jet_ir_length</code>	Specifies the number of IR bits preceding the core to connect to.
<code>--jet_itc_output</code>	Sends text to the <code>stdout</code> and <code>stderr</code> streams by way of trace output, for SiFive devices with an ITC.
<code>--jet_power_from_probe</code>	Specifies the power supply from the I-jet or I-jet Trace probe.
<code>--jet_script_file</code>	Specifies the reset script file.
<code>--jet_sigprobe_opt</code>	Sets the main trace mode for the debug probe.
<code>--jet_standard_reset</code>	Selects the reset strategy to be used when C-SPY starts.
<code>--jet_startup_connection_timeout</code>	Prolongs the time that the C-SPY driver tries to connect to the target board.
<code>--jet_tap_position</code>	Selects a specific device in the JTAG scan chain.
<code>--reset_style</code>	Specifies the reset strategies that will be available when debugging.

OPTIONS AVAILABLE FOR THE C-SPY GDB SERVER DRIVER

`--gdbserv_exec_command` Sends a command string to the GDB Server.

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.

`--application_args`

Syntax

`--application_args="arg0 arg1 ..."`

Parameters

arg

A command line argument.

For use with	cspybat
Description	<p>Use this option to pass command line arguments to the debugged application. These variables must be defined in the application:</p> <pre> /* __argc, the number of arguments in __argv. */ __no_init __root int __argc; /* __argv, an array of pointers to the arguments (strings); must be large enough to fit the number of arguments.*/ __no_init __root const char * __argv[MAX_ARGS]; /* __argvbuf, a storage area for __argv; must be large enough to hold all command line arguments. */ __no_init __root char __argvbuf[MAX_ARG_SIZE]; </pre>
Example	<pre>--application_args="--logfile log.txt --verbose"</pre> <p>To set this option, use Project>Options>Debugger>Extra Options</p>



--attach_to_running_target

Syntax	--attach_to_running_target
For use with	<p>cspybat</p> <p>Note: This option might not be supported by the combination of C-SPY driver and device that you are using. If you are using this option with an unsupported combination, C-SPY produces a message.</p>
Description	<p>Use this option to make the debugger attach to a running application at its current location, without resetting the target system.</p> <p>If you have defined any breakpoints in your project, the C-SPY driver will set them during attachment. If the C-SPY driver cannot set them without stopping the target system, the breakpoints will be disabled. The option also suppresses download and the Run to option.</p>



Project>Attach to Running Target

--backend

Syntax	<code>--backend {driver options}</code>
Parameters	<i>driver options</i> Any option available to the C-SPY driver you are using.
For use with	<code>cspybat</code> (mandatory).
Description	Use this option to send options to the C-SPY driver. All options that follow <code>--backend</code> will be passed to the C-SPY driver, and will not be processed by <code>cspybat</code> itself.
	 This option is not available in the IDE.

--code_coverage_file

Syntax	<code>--code_coverage_file file</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>file</i> The name of the destination file for the code coverage information.
For use with	<code>cspybat</code>
Description	Use this option to enable the generation of a text-based report file for code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file. Because most embedded applications do not terminate, you might have to use this option in combination with <code>--timeout</code> or <code>--cycles</code> . Note that this option requires that the C-SPY driver you are using supports code coverage. If you try to use this option with a C-SPY driver that does not support code coverage, an error message will be directed to <code>stderr</code> .
See also	<i>Code coverage</i> , page 225, <i>--cycles</i> , page 341, <i>--timeout</i> , page 364.
	 To set this option, choose View>Code Coverage , right-click and choose Save As when the C-SPY debugger is running.

--core

Syntax	<pre>--core=RV32{E G I}[M][A][F][D][C][P][N][_Named_ext[_Named_ext1...]] --core=RV64{G I}[M][A][F][D][C][P][N][NamedExt[_NamedExt1...]]</pre>
Parameters	<p>The parameters correspond to the RISC-V extension support. This option reflects the corresponding compiler option. Single letter parameters corresponding to other standard extensions are accepted but ignored.</p>
For use with	All C-SPY drivers.
Description	Use this option to specify the core you are using.
See also	The <i>IAR C/C++ Development Guide for RISC-V</i> for information about the standard extensions.



Project>Options>General Options>Target>Device

--cycles

Syntax	<pre>--cycles <i>cycles</i></pre> <p>Note that this option must be placed before the <code>--backend</code> option on the command line.</p>
Parameters	<p><i>cycles</i></p> <p>The number of cycles to run.</p>
For use with	cspybat
Description	Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted. Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.



This option is not available in the IDE.

--debug_file

Syntax	<code>--debug_file filename</code>
Parameters	<i>filename</i> The name of the debug file to use.
For use with	<code>cspybat</code>
Description	Use this option to make <code>cspybat</code> use the specified debug file instead of the one used in the generated <code>cpsybat.bat</code> file. This option can be placed both before and after the <code>--backend</code> option on the command line.



This option is not available in the IDE.

--device_macro

Syntax	<code>--device_macro filename</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The C-SPY device macro file to be used (filename extension <code>dmac</code>).
For use with	<code>cspybat</code>
Description	Use this option to specify a C-SPY device macro file to be loaded before you execute the target application. A device macro is also loaded when you run a flash loader. A device macro can include scripted reset styles that can be used by the debugger. This option can be used more than once on the command line.
See also	<i>Briefly about using C-SPY macros</i> , page 268.



This option is not available in the IDE.

--disable_interrupts

Syntax	<code>--disable_interrupts</code>
For use with	The C-SPY simulator driver.

Description

Use this option to disable the interrupt simulation.



To set this option, choose **Simulator>Interrupt Configuration** and deselect the **Enable interrupt simulation** command on the context menu.

--disable_misalignment_exception

Syntax

`--disable_misalignment_exception`

For use with

The C-SPY Simulator driver.

Description

By default, the simulator throws an exception when an access to misaligned data is detected. Use this option to disable that behavior and, in effect, allow accesses to misaligned data.



To set this option, choose **Simulator>Data Alignment Setup** and select **Notification: None** or **Notification: Log**.

--download_only

Syntax

`--download_only`

Note that this option must be placed before the `--backend` option on the command line.

For use with

`cspybat`

Description

Use this option to download the code image without starting a debug session afterwards.



Project>Download>Download active application

Alternatively, to set a related option, choose:

Project>Options>Debugger>Setup and deselect **Run to**.

--drv_catch_exceptions

Syntax	<code>--drv_catch_exceptions=<i>value</i></code>	
Parameters	<i>value</i>	<p>A 32-bit value. The lower 28 bits specify which exceptions and interrupts to catch:</p> <ul style="list-style-type: none"> Bit 0 = Reset Bit 1 = Non-maskable interrupt (NMI) Bit 2 = Instruction access misaligned (IAM) Bit 3 = Instruction access fault (IAF) Bit 4 = Illegal instruction (II) Bit 5 = Load access maligned (LAM) Bit 6 = Load access fault (LAF) Bit 7 = Store/AMO access maligned (SAM) Bit 8 = Store/AMO access fault (SAF) Bit 9 = User Mode—Environment call (UEC) Bit 10 = Supervisor Mode—Environment call (SEC) Bit 11 = Machine Mode—Environment call (MEC) Bit 12 = Instruction page fault (IPF) Bit 13 = Load page fault (LPF) Bit 14 = Store/AMO page fault (SPF) Bit 15-23 = Not used Bit 24 = External interrupt Bit 25 = Timer interrupt Bit 26 = Software interrupt Bit 27 = Not used

value (Continued) The upper 4 bits specify in which execution mode (or modes) the exceptions and interrupts will be caught, except for Reset and NMI which, if selected, will be caught regardless of the current exception mode:

Bit 28 = Machine Mode

Bit 29 = Supervisor Mode

Bit 30 = User Mode

Bit 31 = Not used

For use with The C-SPY I-jet driver.

Description Use this option to make the application stop when a certain exception occurs.

Note: Not all exceptions are supported by all devices. Refer to the manufacturer's device manual to confirm which exceptions your device supports.



Project>Options>Debugger>I-jet>Breakpoints>Catch exceptions

--drv_communication

Syntax `--drv_communication=connection`

Parameters Where *connection* is one of these for the C-SPY GDB Server driver:

Via Ethernet `TCPIP:ip_address`

`TCPIP:ip_address,port`

`TCPIP:hostname`

`TCPIP:hostname,port`

Note that if no port is specified, port 3333 is used by default.

Where *connection* is one of these for the I-jet debugger driver:

Via USB port `USB:#serial` where *serial* is a string of digits and letters that identifies which probe you want to connect to. The serial number can be found either printed on the probe, or obtained by connecting a single probe, and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box.

`USB:#select` forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.

For use with Any C-SPY hardware debugger driver.

Description Use this option to choose the communication link.



Project>Options>Debugger>C-SPY driver>Setup.

--drv_communication_log

Syntax `--drv_communication_log=filename`

Parameters *filename* The name of the log file.

For use with All C-SPY hardware debugger 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>C-SPY driver>Log communication

--drv_default_breakpoint

Syntax `--drv_default_breakpoint={0|1|2}`

Parameters 0 Auto (default)
1 Hardware

2 Software

For use with	Any C-SPY hardware debugger driver.
Description	Use this option to select the type of breakpoint resource to be used when setting a breakpoint.
	 Project>Options>Debugger>C-SPY driver>Breakpoints>Default breakpoint type

--drv_exclude_from_verify

Syntax	<code>--drv_exclude_from_verify=startaddr-endaddr</code>	
Parameters	<code>startaddr</code>	The start of the memory range.
	<code>endaddr</code>	The end of the memory range.
For use with	Any C-SPY hardware debugger driver.	
Description	Use this option to exclude memory ranges from being verified when the option <code>--drv_verify_download</code> , or the option Verify download in the Project>Options dialog box in the IDE, is used. The option can be specified multiple times to exclude several ranges.	
	 To set this option, use Project>Options>Debugger>Extra Options .	

--drv_interface

Syntax	<code>--drv_interface={JTAG cJTAG}</code>	
Parameters	JTAG (default)	Specifies the JTAG interface
	cJTAG	Specifies the cJTAG interface.
For use with	The C-SPY I-jet driver.	
Description	Use this option to specify the communication interface between the debug probe and the target system.	



Project>Options>Debugger>I-jet>Interface>Interface

--drv_interface_speed

Syntax	<code>--drv_interface_speed=<i>kHz</i></code>	
Parameters	<i>kHz</i>	The frequency in kHz
For use with	The C-SPY I-jet driver.	
Description	Use this option to set the JTAG communication speed in kHz.	
See also	<i>I-jet : Setup</i> , page 383.	



Project>Options>Debugger>I-jet>Interface>Interface speed

--drv_reset_to_cpu_start

Syntax	<code>--drv_reset_to_cpu_start</code>	
For use with	Any C-SPY hardware debugger driver.	
Description	<p>Normally, at reset, the debugger sets PC to the entry point of the application.</p> <p>This option omits setting the PC each time that the application is reset. This can be useful when you want to keep the reset value that the CPU sets at reset, for example to start executing from the first instruction pointed out by the vector table, or to run a bootloader or OS startup code before entering the start address of the application.</p>	



To set this option, use **Project>Options>Debugger>Extra Options**.

--drv_restore_breakpoints

Syntax	<code>--drv_restore_breakpoints=<i>location</i></code>	
Parameters	<i>location</i>	Address or function name label

For use with	Any C-SPY hardware debugger driver.
Description	Use this option to restore automatically any software breakpoints that were overwritten during system startup.



Project>Options>Debugger>Driver>Breakpoints>Restore software breakpoints at

--drv_suppress_download

Syntax	--suppress_download
For use with	The C-SPY I-jet driver.
Description	Use this option to suppress the downloading of the executable image to a non-volatile type of target memory. The image corresponding to the debugged application must already exist in the target.

If this option is combined with the option `--verify_download`, the debugger will read back the executable image from memory and verify that it is identical to the debugged application.



Project>Options>Debugger>Download>Suppress download

--drv_system_bus_access

Syntax	--drv_system_bus_access
For use with	The C-SPY I-jet driver
Description	Use this option to inform the I-jet driver that the device supports memory access via the system bus. This is required for Live Watch and live updates of the Memory windows.



To set this option, use **Project>Options>Debugger>I-jet>Setup>Allow system bus access**.

--drv_vector_table_base

Syntax	<code>--drv_vector_table_base=expression</code>
Parameters	<i>expression</i> A label or an address
For use with	Any C-SPY hardware debugger driver.
Description	Use this option to specify the location of the reset vector. 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.



Project>Options>Debugger>Extra Options.

--drv_verify_download

Syntax	<code>--verify_download</code>
For use with	Any C-SPY hardware debugger driver.
Description	Use this option to verify that the downloaded code image can be read back from target memory with the correct contents.



Project>Options>Debugger>Download>Verify download

-f

Syntax	<code>-f filename</code>
Parameters	<i>filename</i> A text file that contains the command line options (default filename extension <code>.xcl</code>).
For use with	<code>csybat</code>
Description	Use this option to make <code>csybat</code> read command line options from the specified file.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character is treated like a space or tab character.

Both C/C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

This option can be placed either before or after the `--backend` option on the command line.



To set this option, use **Project>Options>Debugger>Extra Options**.

--flash_loader

Syntax	<code>--flash_loader filename</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The flash loader specification XML file, with the filename extension <code>board</code> .
For use with	<code>cspybat</code>
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 Flash Loader Development Guide</i> .



To set related options, choose:

Project>Options>Debugger>Use flash loader(s)

--function_profiling

Syntax	<code>--function_profiling filename</code>
Parameters	<i>filename</i> The name of the log file where the profiling data is saved.
For use with	The C-SPY simulator driver.

Description Use this option to find the functions in your source code where the most time is spent during execution. The profiling information is saved to the specified file. For more information about function profiling, see *Profiling*, page 215.



C-SPY driver>Function Profiling

--gdbserv_exec_command

Syntax `--gdbserv_exec_command="string"`

Parameters *string* String or command sent to the GDB Server. For more information, see the GDB server documentation.

For use with The C-SPY GDB Server driver.

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



Project>Options>Debugger>Extra Options

--jet_board_cfg

Syntax `--jet_board_cfg=probe_configuration_file`

Parameters *probe_configuration_file* The full path to a probe configuration file.

For use with The C-SPY I-jet driver.

Description Use this option to specify a probe configuration file that defines the debug system on the board.



Project>Options>Debugger>I-jet>Interface>Probe configuration file

--jet_board_did

Syntax	<code>--jet_board_did={cpu #cpu_number}</code>
Parameters	<p><i>cpu</i> If a board configuration file is specified (using <code>--jet_board_cfg</code>) and the defined debug system contains more than one CPU, use this parameter to select a CPU. The value of <i>cpu</i> is a text string. The range of valid values are located in the probe configuration file.</p> <p><i>#cpu_number</i> If the debug system is a JTAG scan chain, and there are several CPUs at the specified TAP position, then specify the CPU number on target.</p> <p>Note that <i>#cpu_number</i> has no effect if a board configuration file is specified using <code>--jet_board_cfg</code>.</p>
For use with	The C-SPY I-jet driver.
Description	Use this option to specify which CPU to debug on a multicore system.
Example	<p>Selecting the CPU on a multicore device with a probe configuration file:</p> <pre>--jet-board-cfg=device.ProbeConfig --jet_board_did=A9_1</pre> <p>Selecting the CPU on a multicore device with a JTAG scan chain, where several CPUs are found at the specified TAP position:</p> <pre>--jet_tap_position=1 --jet_ir_length=5 --jet_board_did=#2</pre> <p> Project>Options>Debugger>I-jet>Interface>Probe configuration file>CPU</p> <p>Project>Options>Debugger>I-jet>Interface>Explicit probe configuration>CPU number on target</p>

--jet_ir_length

Syntax	<code>--jet_ir_length=length</code>
Parameters	<p><i>length</i> The number of IR bits preceding the core to connect to, for JTAG scan chains that mix RISC-V devices with other devices.</p>

For use with	The C-SPY I-jet driver.
Description	Use this option to set the number of IR bits preceding the core to connect to.
See also	<i>I-jet : Interface</i> , page 386



Project>Options>Debugger>I-jet>Interface>Explicit probe configuration>Preceding bits

--jet_itc_output

Syntax	<code>--jet_itc_output</code>
For use with	The C-SPY I-jet driver.
Description	Makes the executing application send text to the <code>stdout</code> and <code>stderr</code> streams by way of trace output, rather than by temporarily stopping the core at a breakpoint. This requires an Instrumentation Trace Component (ITC), present on some SiFive devices.
See also	<i>Trace Settings dialog box</i> , page 180



Project>Options>General Options>Library Configuration>Library low-level interface implementation>Stdout/Stderr>Via SiFive ITC

--jet_power_from_probe

Syntax	<code>--jet_power_from_probe=[leave_on switch_off]</code>	
Parameters	<code>leave_on</code>	Continues to supply power to the target even after the debug session has been stopped.
	<code>switch_off</code>	Turns off the power to the target when the debug session stops.
For use with	The C-SPY I-jet driver.	
Description	Use this option to specify the status of the probe power supply after debugging. If this option is not specified, the probe will not supply power to the board.	



Project>Options>Debugger>I-jet>Setup>Target power

--jet_script_file

Syntax	<code>--jet_script_file=path</code>
Parameters	<i>path</i> The path to the file where the scripted reset strategies are described.
For use with	The C-SPY I-jet driver.
Description	Use this option to specify the file that describes the available scripted reset strategies, if any.
See also	<code>--reset_style</code> , page 362 and <code>--jet_standard_reset</code> , page 357.



To set this option, use **Project>Options>Debugger>Extra Options**.

--jet_sigprobe_opt

Syntax	<code>--jet_sigprobe_opt=trace(mode=<i>behavior</i>[,<i>base</i>=<i>address</i>])</code>
Parameters	<i>behavior</i> The trace behavior when you use the I-jet probe. Choose between: <ul style="list-style-type: none"> <code>auto</code> – Uses External, Serial, or RAM trace, in that order, depending on the capabilities of the debug probe. This is the default trace behavior when using C-SPY without an IDE project. <code>external</code> – Collects External trace data, see <i>External trace</i>, page 176. If this mode is not supported by the debug probe, trace will be disabled for the session. <code>off</code> – This disables I-jet trace completely. Trace cannot be enabled in any trace-related windows for the I-jet driver. This ensures that trace modules will not do any reading/writing to the target system. <code>ram</code> – Collects RAM trace data, see <i>RAM trace</i>, page 176. If this mode is not supported by the debug probe, trace will be disabled for the session. <code>serial</code> – Collects Serial trace data, see <i>Serial trace</i>, page 176. If this mode is not supported by the debug probe, trace will be disabled for the session.

	<i>address</i>	A device-specific base address for trace hardware registers. If no base address is specified, the base address 0x1000'0000 will be used by default.
For use with		The C-SPY I-jet driver and a device that supports trace.
Description		Use this option to change the main trace mode for the debug probe. (This will override the setting in the Trace Settings dialog box in the IDE.)
Example		<code>--jet_sigprobe_opt=trace(mode=serial,base=0x18020000)</code>
See also		<i>Trace Settings dialog box</i> , page 180.
		To set this option in the IDE, use the Mode option in the Trace Settings dialog box.

--jet_standard_reset

Syntax `--jet_standard_reset=strategy,duration,delay`

Parameters

strategy

The reset strategy. Choose between:

- 0, reset disabled
- 1, software reset
- 2, hardware reset
- 3, reset just a single hart
- 4, system reset.

The following reset strategies are available, if present in the file specified by `--jet_script_file` and defined by corresponding instances of `--reset_style`:

- 5, custom reset
- 6, reset by watchdog or reset register
- 7, reset and halt after bootloader
- 8, reset and halt before bootloader
- 9, connect during reset.

For more information, see *I-jet : Setup*, page 383.

duration

The time in milliseconds that the hardware reset asserts the reset signal (line `nSRST/nRESET`) low to reset the device.

Some devices might require a longer reset signal than the default 200 ms.

This parameter applies to the hardware reset, and to those custom reset strategies that use the hardware reset.

	<i>delay</i>	The delay time, in milliseconds, after the reset signal has been de-asserted, before the debugger attempts to control the processor. The processor might be kept internally in reset for some time after the external reset signal has been de-asserted, and therefore inaccessible to the debugger. This parameter applies to the hardware reset, and to those custom reset strategies that use the hardware reset.
For use with	The C-SPY I-jet driver.	
Description	Use this option to select the reset strategy to be used when the debugger starts.	
See also	<i>--reset_style</i> , page 362 and <i>--jet_script_file</i> , page 355.	



Project>Options>Debugger>I-jet>Setup>Reset

--jet_startup_connection_timeout

Syntax	<i>--jet_startup_connection_timeout=milliseconds</i>	
Parameters	<i>milliseconds</i>	The time in milliseconds.
For use with	The C-SPY I-jet driver.	
Description	Use this option to prolong the time that the C-SPY driver tries to connect to the target board.	



To set this option, use **Project>Options>Debugger>Extra Options**.

--jet_tap_position

Syntax	<i>--jet_tap_position=tap_number multidrop_id</i>	
Parameters	<i>tap_number</i>	The TAP position of the device you want to connect to.
	<i>multidrop_id</i>	The target ID in a multi-drop system.

For use with	The C-SPY I-jet 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>I-jet</i> : <i>Interface</i> , page 386.
	 Project>Options>Debugger>I-jet>Interface>Explicit probe configuration>Target number (TAP or target ID)

--leave_target_running

Syntax	<code>--leave_target_running</code>
For use with	<code>cspybat</code> . Any C-SPY hardware debugger driver. Note: Even if this option is supported by the C-SPY driver you are using, there might be device-specific limitations.
Description	Use this option to make the debugger leave the application running on the target hardware after the debug session is closed.
	 Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this option.
	 C-SPY driver>Leave Target Running

--macro

Syntax	<code>--macro filename</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The C-SPY macro file to be used (filename extension <code>mac</code>).
For use with	<code>cspybat</code>
Description	Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.

See also

Briefly about using C-SPY macros, page 268.**Project>Options>Debugger>Setup>Setup macros>Use macro file**

--macro_param

Syntax

`--macro_param [param=value]`Note that this option must be placed before the `--backend` option on the command line.

Parameters

*param=value**param* is a parameter defined using the `__param` C-SPY macro construction.
value is a value.

For use with

`cspybat`

Description

Use this option to assign a value to a C-SPY macro parameter. This option can be used more than once on the command line.

See also

Macro parameters, page 275.To set this option, use **Project>Options>Debugger>Extra Options**

--mapu

Syntax

`--mapu`

For use with

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 `stderr` and the execution will stop.

See also

Monitoring memory and registers, page 133.

To set related options, choose:

Simulator>Memory Access Setup

--multicore_nr_of_cores

Syntax	<code>--multicore_nr_of_cores=cores</code>
Parameters	<i>cores</i> The number of cores on your device. This must be an integer from 2–8.
For use with	The C-SPY simulator driver.
Description	For symmetric multicore debugging, specify the number of cores on your device. This option is not needed for debugging a single-core system, or for asymmetric multicore debugging.
See also	<i>Multicore debugging</i> , page 233.



Project>Options>Debugger>Multicore>Number of cores

-p

Syntax	<code>-p filename</code>
Parameters	<i>filename</i> The device description file to be used.
For use with	All C-SPY drivers.
Description	Use this option to specify the device description file to be used.
See also	<i>Selecting a device description file</i> , page 43.



Project>Options>Debugger>Setup>Device description file

--plugin

Syntax	<code>--plugin filename</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The plugin file to be used (filename extension <code>dll</code>).

For use with	<code>cspybat</code>
Description	<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>riscvbat.dll</code> located in the <code>riscv\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 also include 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>



Project>Options>Debugger>Plugins

--reset_style

Syntax	<code>--reset_style="reset_id, reset_name, selected, menu_command"</code>								
Parameters	<table> <tr> <td><code>reset_id</code></td> <td>The number of the reset strategy, 0-9, as described for <code>--jet_standard_reset</code></td> </tr> <tr> <td><code>reset_name</code></td> <td>The name of the reset strategy, according to the file specified by <code>--jet_script_file</code>. For the built-in reset strategies, this parameter is -. To override a built-in reset strategy, enter the label or function name in your reset script file.</td> </tr> <tr> <td><code>selected</code></td> <td>0 or 1, where 1 sets the default reset strategy for the Reset drop-down button</td> </tr> <tr> <td><code>menu_command</code></td> <td>The name of the reset strategy as it will be displayed on the Reset drop-down menu.</td> </tr> </table>	<code>reset_id</code>	The number of the reset strategy, 0-9, as described for <code>--jet_standard_reset</code>	<code>reset_name</code>	The name of the reset strategy, according to the file specified by <code>--jet_script_file</code> . For the built-in reset strategies, this parameter is -. To override a built-in reset strategy, enter the label or function name in your reset script file.	<code>selected</code>	0 or 1, where 1 sets the default reset strategy for the Reset drop-down button	<code>menu_command</code>	The name of the reset strategy as it will be displayed on the Reset drop-down menu.
<code>reset_id</code>	The number of the reset strategy, 0-9, as described for <code>--jet_standard_reset</code>								
<code>reset_name</code>	The name of the reset strategy, according to the file specified by <code>--jet_script_file</code> . For the built-in reset strategies, this parameter is -. To override a built-in reset strategy, enter the label or function name in your reset script file.								
<code>selected</code>	0 or 1, where 1 sets the default reset strategy for the Reset drop-down button								
<code>menu_command</code>	The name of the reset strategy as it will be displayed on the Reset drop-down menu.								
For use with	The C-SPY I-jet driver.								
Description	Use this option to specify the reset strategies that will be available when debugging, once for each reset strategy.								

Example This example specifies a script file, sets the standard reset strategy, and specifies the reset strategies that will be available when debugging:

```
--jet_script_file=myDir\myProbeScriptFile
--jet_standard_reset=9,0,0
--reset_style="0,-,0,Disabled (no reset)"
--reset_style="1,-,0,Software"
--reset_style="2,-,0,Hardware"
--reset_style="3,-,0,Core"
--reset_style="4,-,0,System"
--reset_style="5,Custom,0,Custom reset"
--reset_style="9,ConnectUnderReset,1,Connect during reset"
```

See also `--jet_script_file`, page 355 and `--jet_standard_reset`, page 357



To set this option, use **Project>Options>Debugger>Extra Options**.

--silent

Syntax `--silent`

Note that this option must be placed before the `--backend` option on the command line.

For use with `cspybat`

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



This option is not available in the IDE.

--suppress_entrypoint_warning

Syntax `--suppress_entrypoint_warning`

For use with `cspybat`

Description Use this option to disable the warning in the debug log when a debug session starts with the ELF entry point at address `0x0` (which according to the ELF standard means that there is no entry point). In practice, there is no problem with having the entry point at address `0x0`.



To set this option, use **Project>Options>Debugger>Extra Options**

--timeout

Syntax

`--timeout milliseconds`

Note that this option must be placed before the `--backend` option on the command line.

Parameters

milliseconds

The number of milliseconds before the execution stops.

For use with

`cspybat`

Description

Use this option to limit the maximum allowed execution time.



This option is not available in the IDE.

Flash loaders

- Introduction to the flash loader
- Using flash loaders
- Reference information on the flash loader

Introduction to 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.

Flash loaders for various microcontrollers are provided with IAR Embedded Workbench for RISC-V. 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.

Using flash loaders

These tasks are covered:

- Setting up the flash loader(s)
- Aborting a 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.

ABORTING A FLASH LOADER

To abort a flash loader:

- 1 Press Ctrl+Shift+. (period) for a short while.
- 2 A message that says that the flash loader has aborted is displayed in the **Debug Log** window.

This method can be used if you suspect that something is wrong with the execution, for example because it seems not to terminate in a reasonable time.

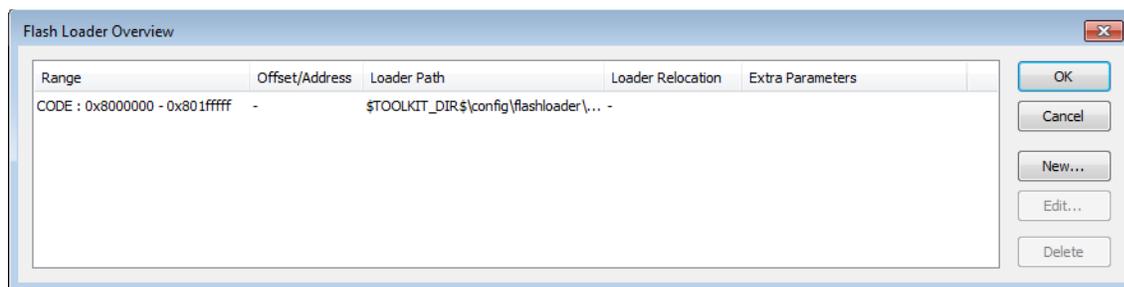
Reference information on the flash loader

Reference information about:

- *Flash Loader Overview dialog box*, page 366
- *Flash Loader Configuration dialog box*, page 368

Flash Loader Overview dialog box

The **Flash Loader Overview** dialog box is available from the **Project>Options>Debugger>Download** page.



This dialog box lists all defined flash loaders. If you have selected a device on the **Project>Options>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.

Requirements

Available for supported hardware debugger systems.

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 by an **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).

Loader Relocation

For relocatable flash loaders, this is the start of the target RAM memory where the flash loader will be downloaded.

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 *Flash Loader Configuration dialog box*, page 368.

Edit

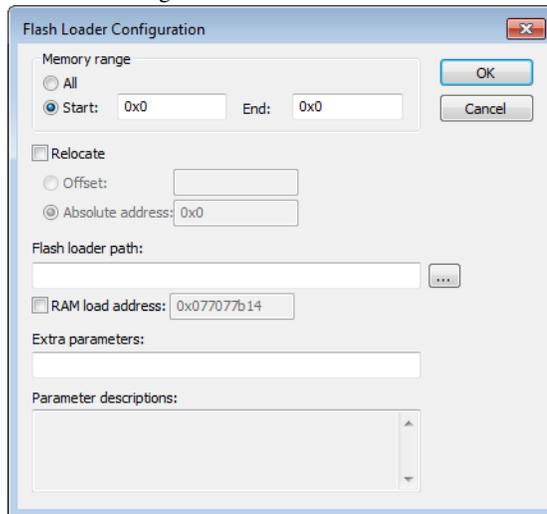
Displays a dialog box where you can modify the settings for the selected flash loader, see *Flash Loader Configuration dialog box*, page 368.

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.



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.

Requirements

Available for supported hardware debugger systems.

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, in other words, relocates 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.

RAM load address

If the flash loader is relocatable, this option overrides the default address in the target RAM memory that flash loader is downloaded to, in other words, relocates the flash loader. Use the text box to specify the address.

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.

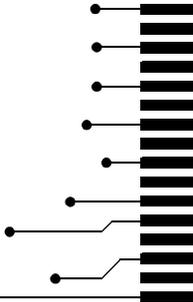
Parameter descriptions

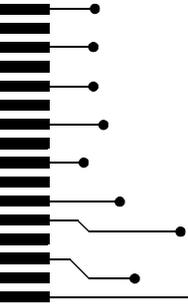
Displays a description of the extra parameters specified in the **Extra parameters** text box.

Part 4. Additional reference information

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

- Debugger options
- Additional information on C-SPY drivers





Debugger options

- Setting debugger options
- Reference information on general debugger options
- Reference information on C-SPY hardware debugger driver options

Setting debugger options

Before you start the C-SPY debugger you might need to set some options—both C-SPY generic options and options required for the target system (C-SPY driver-specific options).

To set debugger options in the IDE:

- 1 Choose **Project>Options** to display the **Options** dialog box.
- 2 Select **Debugger** in the **Category** list.

For more information about the generic options, see *Reference information on general debugger options*, page 374.

- 3 On the **Setup** page, make sure to select the appropriate C-SPY driver from the **Driver** drop-down list.
- 4 To set the driver-specific options, select the appropriate driver from the **Category** list. Depending on which C-SPY driver you are using, different options are available.

C-SPY driver	Available options pages
GDB Server driver	<i>GDB Server: Setup</i> , page 381 <i>GDB Server: Breakpoints</i> , page 382
C-SPY I-Jet driver	<i>I-jet : Setup</i> , page 383 <i>I-jet : Interface</i> , page 386 <i>I-jet : Breakpoints</i> , page 388
Third-party driver	<i>Third-Party Driver options</i> , page 389

Table 36: Options specific to the C-SPY drivers you are using

- 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 general debugger options

Reference information about:

- *Setup*, page 374
- *Download*, page 375
- *Images*, page 376
- *Multicore*, page 377
- *Extra Options*, page 379
- *Plugins*, page 380

Setup

The general **Setup** options select the C-SPY driver, the setup macro file, and device description file to use, and specify which default source code location to run to.

Driver

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

Run to

Specifies the location C-SPY runs to when the debugger starts after a reset. By default, C-SPY runs to the `main` function.

To override the default location, specify the name of a different location you want C-SPY to run to. You can specify assembler labels or whatever can be evaluated as such, for example function names.

If the option is deselected, the program counter will contain the regular hardware reset address at each reset.

See also *Executing from reset*, page 42.

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.

Device description file

A default device description file is selected automatically based on your project settings. To override the default file, select **Override default** and specify an alternative file. A browse button is available.

For information about the device description file, see *Modifying a device description file*, page 47.

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.

The screenshot shows a dialog box titled "Download" with the following options:

- Verify download
- Suppress download
- Use flash loader(s)
- Override default board file

Below the "Override default board file" checkbox is a text input field and a browse button "...". At the bottom left of the dialog is an "Edit..." button.

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.

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.

See *Flash loaders*, page 365.

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.

Images

The **Images** options control the use of additional debug files to be downloaded.

The screenshot shows a configuration window titled "Images". It contains three identical-looking sections for downloading extra images. The first section is checked with a checkbox and includes a "Path:" text box with a browse button "...", an "Offset:" text box, and a "Debug info only" checkbox. The second and third sections are unchecked and also include "Path:" and "Offset:" text boxes with browse buttons, and a "Debug info only" checkbox.

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.

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 debug images, use the related C-SPY macro, see `__loadImage`, page 297.

For more information, see *Loading multiple debug images*, page 45.

Multicore

The **Multicore** options configure multicore debugging.

Number of cores

For symmetric multicore debugging, specify the number of cores on your device. For asymmetric multicore debugging, specify the number of cores in the master project. (Normally, this will be 1.)

Disabled

Selecting this option makes the debug session symmetric multicore.

Simple

Selecting this option makes the debug session an asymmetric multicore debugger master. When you start a debug session, a new instance of the IAR Embedded Workbench IDE will be started, using the following options:

Partner workspace

Specify the workspace to be opened in the partner (slave) instance.

Partner project

Specify the name of the project in the workspace to be opened in the partner instance. For example, if the project filename is `MyPartnerProj.ewp`, specify `MyPartnerProj`.

Partner configuration

Specify the build configuration to be used when debugging the partner. For example, `Debug` or `Release`.

Attach partner to running target

If you have selected the command **Attach to Running Target** from the **Project** menu, which affects the master. You can also select **Attach partner to running target** to also make the debugger attach the partner to the running application at its current location, without resetting the target system.

For information about **Attach to Running Target**, see the *IDE Project Management and Building Guide for RISC-V*.

Partner cores

Specify the number of cores in the partner project.

Override partner debugger location

If the Embedded Workbench instance associated with the *partner* project is not installed in the same location as the Embedded Workbench instance associated with the *master* project, for example in `c:\Program Files\IAR Systems\Embedded Workbench N.n.`, you must specify the installation directory of the Embedded Workbench for the partner project. Note that the Embedded Workbench must be based on version 9.1.7 or later of the shared components—to check this, choose **Help>About>Product Info**.

Advanced

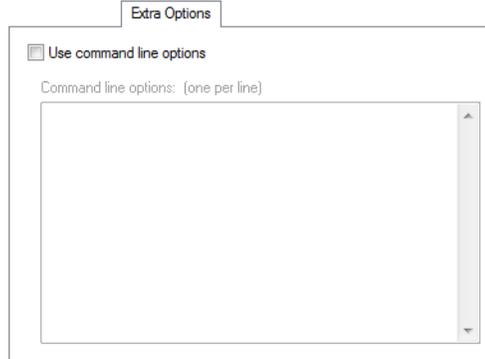
Selecting this option makes the debug session asymmetric multicore with one or more new instances of the IAR Embedded Workbench IDE.

Session configuration

Use the browse button to specify the XML multicore session file that contains the settings for the debug session. For more information about this file, see *The multicore session file*, page 241.

Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.



Use command line options

Specify command line arguments that are not supported by the IDE to be passed to C-SPY.

Note that it is possible to use the `/args` option to pass command line arguments to the debugged application.

The syntax is:

```
/args arg0 arg1 ...
```

Multiple lines with `/args` are allowed, for example:

```
/args --logfile log.txt
```

```
/args --verbose
```

If you use `/args`, these variables must be defined in your application:

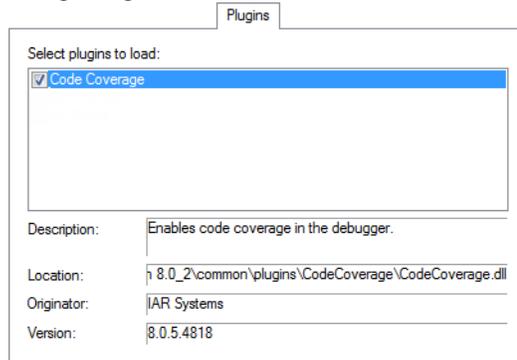
```
/* __argc, the number of arguments in __argv. */
__no_init __root int __argc;
```

```
/* __argv, an array of pointers to strings that holds the
arguments; must be large enough to fit the number of
parameters.*/
__no_init __root const char * __argv[MAX_ARGS];
```

```
/* __argvbuf, a storage area for __argv; must be large enough to
hold all command line parameters. */
__no_init __root char __argvbuf[MAX_ARG_SIZE];
```

Plugins

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



Select plugins to load

Selects the plugin modules to be loaded and made available during debug sessions. The list contains the plugin modules delivered with the product installation.

Description

Describes the plugin module.

Location

Informs about the location of the plugin module.

Generic plugin modules are stored in the `common\plugins` directory. Target-specific plugin modules are stored in the `riscv\plugins` directory.

Originator

Informs about the originator of the plugin module, which can be modules provided by IAR or by third-party vendors.

Version

Informs about the version number.

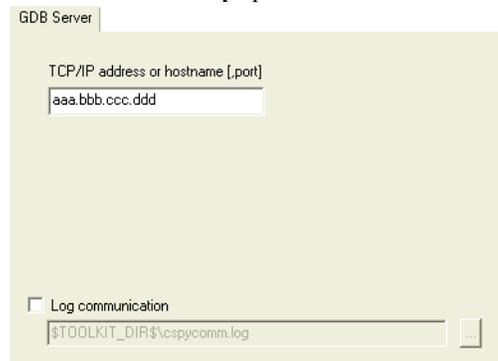
Reference information on C-SPY hardware debugger driver options

Reference information about:

- *GDB Server: Setup*, page 381
- *GDB Server: Breakpoints*, page 382
- *I-jet : Setup*, page 383
- *I-jet : Interface*, page 386
- *I-jet : Breakpoints*, page 388
- *Third-Party Driver options*, page 389

GDB Server: Setup

The GDB Server **Setup** options control the C-SPY GDB Server driver.



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, detailed knowledge of the interface is required.

GDB Server: Breakpoints

The **Breakpoints** options specify the breakpoint behavior for the C-SPY GDB Server driver.

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. |
| 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

Restores software breakpoints that were overwritten 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.

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`.

I-jet : Setup

The **Setup** options control the I-jet in-circuit debugging probes:

Reset

Selects the reset strategy to be used when the debugger starts. Choose between:

Disabled (no reset)

No reset is performed.

Software

Sets `PC` to the program entry address and `SP` to the initial stack pointer value.

This is a software reset.

Hardware

The probe toggles the `nSRST/nRESET` line on the JTAG connector to reset the device. Usually, this reset also resets the peripheral units.

The processor should stop at the reset handler before executing any instruction. Some processors might not stop at the reset vector, but will be halted soon after, executing some instructions.

Core

Resets a single hart but any peripheral units are not affected. Only available for some devices.

System

Resets all harts of the processor and all peripheral units. Reset vector catch is used for stopping the CPU at the reset vector before the first instruction is executed. Only available for some devices.

All strategies halt the CPU after the reset.

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 to its reset state.

Normally, a C-SPY reset is only a software reset. If you use the **Hardware** 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.



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.

Duration

The time in milliseconds that the hardware reset asserts the reset signal (line `nSRST/nRESET`) low to reset the device.

Some devices might require a longer reset signal than the default 300 ms.

This option applies to the hardware reset, and to those custom reset strategies that use the hardware reset.

Override default delay after reset

Use this option to change the default delay time, in milliseconds, after the reset signal has been de-asserted, before the debugger attempts to control the processor.

The processor might be kept internally in reset for some time after the external reset signal has been de-asserted, and thus inaccessible for the debugger.

This option applies to the hardware reset, and to those custom reset styles that use the hardware reset.

Target power

These options specify how the target system is powered, and if it is powered from the debug probe, the status of the power supply after debugging.

From the probe

Supplies the target system with power from the probe. If this option is deselected, the target system must be powered separately.

Leave on after debugging

Continues to supply the target system with power from the probe even after the debug session has been stopped.

Switch off after debugging

Turns off the power to the target when the debug session stops.

Emulator

These options are used for identifying the debug probe to use.

Always prompt for probe selection

Makes C-SPY always ask you to confirm which probe to use, if more than one debug probe is connected to the host computer.

Serial no

Enter the serial number of the debug probe you are using.

Allow system bus access

Informs the C-SPY driver that the target device supports memory access via the system bus. This is required for Live Watch and live updates of the Memory windows.

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.

I-jet : Interface

The **Interface** options specify the interface between I-jet and the target system.

Probe config

Auto

The I-jet driver automatically identifies the target CPU. It uses the default probe configuration file, if there is one.

This works best if there is only one CPU present.

From file

Specifies that the probe configuration file needs to be overridden, or that there are several target CPUs.

Explicit

Specify how to find the target CPU.

Interface

Selects the communication interface between the debug probe and the target system. Choose between:

JTAG

Uses the JTAG interface.

cJTAG

Uses the cJTAG interface.

Interface speed

Specify the JTAG communication speed. Choose between:

Auto detect

Automatically uses the highest possible frequency for reliable operation.

***n* Hz**

If there are JTAG communication problems or problems in writing to target memory (for example during program download), these problems might be resolved if the speed is set to a lower frequency.

Probe configuration file**Override default**

Specify a probe configuration file to be used instead of the default probe configuration file that comes with the product package.

CPU

Specify the target CPU.

Explicit probe configuration**Multi-target debug system**

Select this option to enable the other options in this section. This might be needed if the automatic probe configuration does not work, regardless of the number of cores in the target system and the number of TAPs on the JTAG chain.

Target number (TAP or target ID)

If the debug system is a JTAG scan chain, specify the **Target number TAP** (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero. If there are several cores at the TAP position, you also need to specify the **CPU number on target**.

Target with multiple CPUs

If you have multiple cores or harts (hardware threads) on the selected TAP, you can select this option to specify which one you want to debug.

CPU number on target

Specify the index number of the core/hart on selected TAP. Hart numbers start from 0.

JTAG scan chain contains non-RISC-V devices

Enables JTAG scan chains that mix RISC-V devices with other devices.

Preceding bits

Specify the TAP (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero.

I-jet : Breakpoints

The **Breakpoints** options specify the breakpoint behavior for I-jet.

The screenshot shows a dialog box titled "Breakpoints". It contains two main sections:

- Default breakpoint type:** Three radio buttons are present: "Auto" (selected), "Hardware", and "Software".
- Catch exceptions:** A large rectangular area containing a grid of checkboxes.
 - Top row: "Reset" (unchecked), "NMI" (unchecked).
 - Second row: "Machine Mode" (checked), "Supervisor Mode" (checked), "User Mode" (checked).
 - Third row: "Instruction access misaligned" (unchecked), "Environment call" (unchecked).
 - Fourth row: "Instruction access fault" (unchecked), "Instruction page fault" (unchecked).
 - Fifth row: "Illegal instruction" (unchecked), "Load page fault" (unchecked).
 - Sixth row: "Load access misaligned" (unchecked), "Store/AMO page fault" (unchecked).
 - Seventh row: "Load access fault" (unchecked), "External interrupt" (unchecked).
 - Eighth row: "Store/AMO access misaligned" (unchecked), "Timer interrupt" (unchecked).
 - Ninth row: "Store/AMO access fault" (unchecked), "Software interrupt" (unchecked).

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.
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.

Catch exceptions

Sets a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. The settings you make will work as default settings for the project and are preserved during debug sessions.

Reset	Catches reset exceptions.
NMI	Catches non-maskable interrupts.

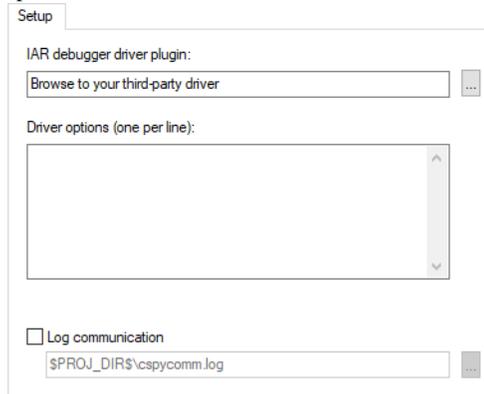
Select which of the exceptions or interrupts listed in the lower half of the group box to catch, and in which execution modes (**Machine Mode**, **Supervisor Mode**, and/or **User Mode**) they will be caught. If you do not select at least one execution mode, none of the

exceptions listed below the mode options will be caught, even if they are selected. By default, all three mode options are selected.

Note: Not all exceptions are supported by all devices. Refer to the manufacturer's device manual to confirm which exceptions your device supports.

Third-Party Driver options

The **Third-Party Driver** options are used for loading any driver plugin provided by a third-party vendor. These drivers must be compatible with the C-SPY debugger driver specification.



IAR debugger driver plugin

Specify the file path to the third-party driver plugin DLL file. A browse button is available for your convenience.

Driver options

This box provides you with a command line interface to C-SPY. Use it to set options for 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.

Additional information on C-SPY drivers

This chapter describes the additional menus and features provided by the C-SPY® drivers. You will also find some useful hints about resolving problems.

Reference information on C-SPY driver menus

Reference information about:

- *C-SPY driver*, page 391
- *Simulator menu*, page 392
- *GDB Server menu*, page 396
- *I-jet menu*, page 396

C-SPY driver

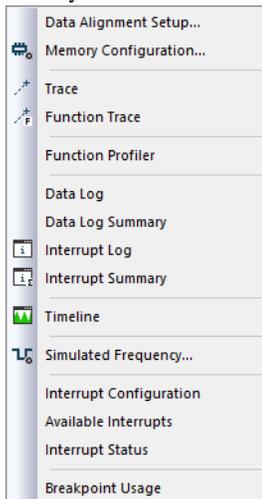
Before you start the C-SPY debugger, you must first specify a C-SPY driver in the **Options** dialog box, using the option **Debugger>Setup>Driver**.

When you start a debug session, a menu specific to that C-SPY driver will appear on the menu bar, with commands specific to the driver.

When we in this guide write “choose *C-SPY driver*>” followed by a menu command, *C-SPY driver* refers to the menu. If the feature is supported by the driver, the command will be on the menu.

Simulator menu

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



Menu commands

These commands are available on the menu:

Data Alignment Setup

Displays a dialog box to control the behavior of the simulator when misaligned data accesses are detected, see *Data Alignment Setup dialog box*, page 394.



Memory Configuration

Displays a dialog box where you configure C-SPY to match the memory of your device, see *Memory Configuration dialog box for the C-SPY simulator*, page 162.



Trace

Opens a window which displays the collected trace data, see *Trace window*, page 183.



Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 190.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 220.

Data Log

Opens a window which logs accesses to up to four different memory locations or areas, see *Data Log window*, page 202.

Data Log Summary

Opens a window which displays a summary of data accesses to specific memory location or areas, see *Data Log Summary window*, page 205.

**Interrupt Log**

Opens a window which displays the status of all defined interrupts, see *Interrupt Log window*, page 257.

**Interrupt Log Summary**

Opens a window which displays a summary of the status of all defined interrupts, see *Interrupt Log Summary window*, page 260.

**Timeline**

Opens a window which gives a graphical view of various kinds of information on a timeline, see *The application timeline*, page 197.

**Simulated Frequency**

Opens the **Simulated Frequency** dialog box where you can specify the simulator frequency used when the simulator displays time information, for example in the log windows. Note that this does not affect the speed of the simulator. For more information, see *Simulated Frequency dialog box*, page 395.

Interrupt Configuration

Opens a window where you can configure C-SPY interrupt simulation, see *Interrupt Configuration window*, page 251.

Available Interrupts

Opens a window with an overview of all available interrupts. You can also force an interrupt instantly from this window, see *Available Interrupts window*, page 254.

Interrupt Status

Opens a window from where you can instantly trigger an interrupt, see *Interrupt Status window*, page 255.

**Breakpoint Usage**

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 117.

Reference information on the C-SPY simulator

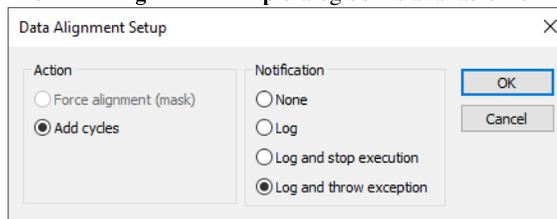
This section gives additional reference information on the C-SPY simulator, and reference information not provided elsewhere in this documentation.

Reference information about:

- *Data Alignment Setup dialog box*, page 394
- *Simulated Frequency dialog box*, page 395

Data Alignment Setup dialog box

The **Data Alignment Setup** dialog box is available from the **Simulator** menu.



Use this dialog box to control the behavior of the simulator when an access to misaligned data is detected.

Note: The hardware of your specific core might not support all actions.

Requirements

The C-SPY simulator.

Action

Selects the action to take when an access to misaligned data is detected. Choose between:

Force alignment

Forces misaligned data to a correct alignment.

Add cycles

Adds one extra bus cycle.

Notification

Selects a suitable notification method. Choose between:

None

Specifies that no notification will be issued.

Log

Displays a message in the **Debug Log** window.

Log and stop execution

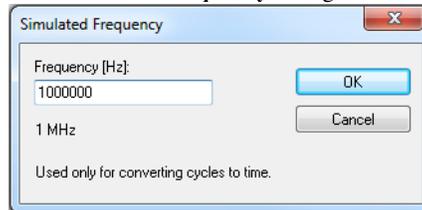
Displays a message in the **Debug Log** window and stops the execution after the access to the misaligned data.

Log and throw exception

Displays a message in the **Debug Log** window and throws an exception after the access to the misaligned data.

Simulated Frequency dialog box

The **Simulated Frequency** dialog box is available from the C-SPY driver menu.



Use this dialog box to specify the simulator frequency used when the simulator displays time information.

Requirements

The C-SPY simulator.

Frequency

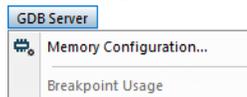
Specify the frequency in Hz.

Reference information on the C-SPY hardware debugger drivers

This section gives additional reference information on the C-SPY hardware debugger drivers, reference information not provided elsewhere in this documentation.

GDB Server menu

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



Menu commands

These commands are available on the menu:

Memory Configuration

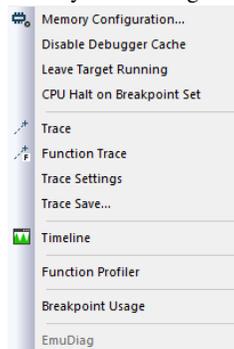
Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers*, page 166.

Breakpoint Usage

Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 117.

I-jet menu

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



Menu commands

These commands are available on the menu:

Memory Configuration

Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers*, page 166.

Disable Debugger Cache

Disables memory caching and memory range checking in C-SPY.

Normally, C-SPY uses the memory range information in the **Memory Configuration** dialog box both to restrict access to certain parts of target memory and to cache target memory contents for improved C-SPY performance. Under certain rare circumstances, this is not appropriate, and you can choose **Disable Debugger Cache** to turn off the caching and memory range checking completely. All accesses from C-SPY will then result in corresponding accesses to the target system. Some of those circumstances are:

- When memory is remapped at runtime and cannot be specified as a fixed set of ranges.
- When the memory range setup is incorrect or incomplete.

Leave Target Running

Leaves the application running on the target hardware after the debug session is closed.

Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.

CPU Halt on Breakpoint Set

Makes it possible to set a breakpoint in an executing application on hardware that does not support setting breakpoints while running. Setting a breakpoint halts the core, sets the breakpoint, and starts the core again.

Trace

Opens the **Trace** window, see *Trace window*, page 183.

Function Trace

Opens the **Function Trace** window, see *Function Trace window*, page 190.

Trace Settings

Displays a dialog box, see *Trace Settings dialog box*, page 180.

Trace Save

Displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.

Timeline

Opens a window, see *Reference information on application timeline*, page 201.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 220.

Breakpoint Usage

Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 117.

EmuDiag

Starts the **EmuDiag** application where you can diagnose the connection between the host computer, the probe, and the board.

Resolving problems

These topics are covered:

- Write failure during load
- No contact with the target hardware

Debugging using the C-SPY hardware debugger systems requires interaction between many systems, independent from each other. For this reason, setting up this debug system can be a complex task. If something goes wrong, it might be difficult to locate the cause of the problem.

This section includes suggestions for resolving the most common problems that can occur when debugging with the C-SPY hardware debugger systems.

For problems concerning the operation of the evaluation board, refer to the documentation supplied with it, or contact your hardware distributor.

WRITE FAILURE DURING LOAD

There are several possible reasons for write failure during load. The most common is that your application has been incorrectly linked:

- Check the contents of your linker configuration file and make sure that your application has not been linked to the wrong address
- Check that you are using the correct linker configuration file.



In the IDE, the linker configuration file is automatically selected based on your choice of device.

To choose a device:

- 1** Choose **Project>Options**.
- 2** Select the **General Options** category.
- 3** Click the **Target** tab.
- 4** Choose the appropriate device from the **Device** drop-down list.

To override the default linker configuration file:

- 1** Choose **Project>Options**.
- 2** Select the **Linker** category.
- 3** Click the **Config** tab.
- 4** Select the **Override default** option, and choose the appropriate linker configuration file in the **Linker configuration file** area. A browse button is available.

NO CONTACT WITH THE TARGET HARDWARE

There are several possible reasons for C-SPY to fail to establish contact with the target hardware. Do this:

- Check the communication devices on your host computer
- Verify that the cable is properly plugged in and not damaged or of the wrong type
- Make sure that the evaluation board is supplied with sufficient power
- Check that the correct options for communication have been specified in the IAR Embedded Workbench IDE.

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

A

Abort (Report Assert option) 75
 __abortLaunch (C-SPY system macro) 286
 absolute location, specifying for a breakpoint 127
 Access type (Edit Memory Access option) 165
 Access (Edit SFR option) 161
 Action (Data Alignment Setup option) 394
 Add Interrupt (Interrupt Configuration option) 253
 Address Range (Find in Trace option) 195
 Address (Edit SFR option) 161
 Advanced (multicore debugger option) 378
 alignment
 allowing misaligned data
 in IDE 394
 on command line 343
 detecting misaligned data 394
 forcing misaligned data 394
 Ambiguous symbol (Resolve Symbol Ambiguity option) . 103
 --application_args (C-SPY command line option) 338
 application, built outside the IDE 44
 __argCount (C-SPY system macro) 283
 assembler labels, viewing 82
 assembler source code, fine-tuning 215
 assembler symbols, in C-SPY expressions 79
 assembler variables, viewing 82
 assumptions, programming experience 21
 Auto Scroll
 (Timeline window context menu) 209, 264
 Auto window 83
 Autostep settings dialog box 75
 Available Interrupts window 254

B

--attach_to_running_target
 (C-SPY command line option) 339
 --backend (C-SPY command line option) 340
 backtrace information, viewing in Call Stack window . . 70

batch mode, using C-SPY in 333
 Big Endian (Memory window context menu) 141
 blocks, in C-SPY macros 277
 bold style, in this guide 25
 breakpoint condition, example 113–114
 breakpoint dialog box
 Code 118
 Data 121, 123
 Data Log 124
 Immediate 125
 Log 120
 Trace Start Trigger 191
 Trace Stop Trigger 193
 Breakpoint Usage window 117
 breakpoints
 briefly about 105
 code, example 310
 connecting a C-SPY macro 272
 consumers of 109
 data 121, 123
 data log 124
 description of 105
 disabling used by Stack window 109
 icons for in the IDE 108
 in Memory window 112
 listing all 117
 reasons for using 105
 setting
 in memory window 112
 using system macros 112
 using the dialog box 110
 single-stepping if not available 42
 toggling 110
 types of 106
 useful tips 113
 Breakpoints window 115
 Browse (Trace toolbar) 184
 byte order, setting in Memory window 140
 __bytes2Word16 (C-SPY system macro) 283

`__bytes2Word32` (C-SPY system macro) 283

C

C function information, in C-SPY 63
C symbols, in C-SPY expressions 78
C variables, in C-SPY expressions 78
Cache type (Edit Memory Range option) 170
call chain, displaying in C-SPY 63
Call Stack graph (Timeline window) 207
Call stack information 63
Call Stack window 70
 for backtrace information 63
Call Stack (Timeline window context menu) 210
`__cancelAllInterrupts` (C-SPY system macro) 287
`__cancelInterrupt` (C-SPY system macro) 287
Catch exceptions (I-jet option) 388
Clear Group
(Registers User Groups Setup window context menu) . . . 156
Clear trace data (Trace toolbar) 183
`__clearBreak` (C-SPY system macro) 287
clock frequency, simulated 395
`__closeFile` (C-SPY system macro) 288
code breakpoints
 overview 106
 toggling 110
code coverage
 real-time 67, 225
 using 226
Code Coverage window 227
`--code_coverage_file` (C-SPY command line option) . . . 340
code, covering execution of 227
command line options 338
 typographic convention 25
command prompt icon, in this guide 25
computer style (monospace font), typographic convention . 25
conditional statements, in C-SPY macros 276
context menu, in windows 81
conventions, used in this guide 24

Copy Window Contents
(Disassembly window context menu) 69
copyright notice 2
Core (Cores window) 239
`--core` (C-SPY command line option) 341
cores
 debugging multiple 233
 inspecting state of 239
Cores window 239
CPU number on target (I-jet setting) 387
`csybat` 333
 reading options from file (-f) 350
current position, in C-SPY Disassembly window 66
cursor, in C-SPY Disassembly window 66
`--cycles` (C-SPY command line option) 341
Cycles (Cores window) 240
C-SPY
 batch mode, using in 333
 debugger systems, overview of 33
 environment overview 29
 plugin modules, loading 43
 scripting. *See* macros
 setting up 41–42
 starting the debugger 43
C-SPY drivers
 differences between drivers 35
 overview 35
 specifying 374
 types of 34
C-SPY expressions 78
 evaluating, using Macro Quicklaunch window 331
 evaluating, using Quick Watch window 97
 in C-SPY macros 276
 Tooltip watch, using 77
 Watch window, using 77
C-SPY hardware drivers, hardware installation 38
C-SPY macros
 blocks 277
 conditional statements 276
 C-SPY expressions 276

- examples 269
 - checking status of register 271
 - creating a log macro 272
 - executing 269
 - connecting to a breakpoint 272
 - using Quick Watch 271
 - using setup macro and setup file 271
 - functions 79, 274
 - keywords 274–275, 277
 - loop statements 277
 - macro statements 276
 - parameters 275
 - setup macro file 268
 - executing 271
 - setup macro functions 268
 - summary 279
 - system macros, summary of 283
 - using 267
 - variables 80, 274
- C-SPY options
- Extra Options 379
 - Images 376
 - Multicore 377
 - Plugins 380
 - Setup 374
- C-SPYLink 34
- C-STAT for static analysis, documentation for 24
- C++ exceptions
- debugging 51–52
 - single stepping 58
- ## D
- Data Alignment Setup dialog box 394
 - data breakpoints, overview 106
 - Data Coverage (Memory window context menu) 141
 - data coverage, in Memory window 139
 - data log breakpoints, overview 107
 - Data Log Summary window 205
 - Data Log window 202
 - ddf (filename extension), selecting a file 43
 - Debug Log window 73
 - Debug menu (C-SPY main window) 50
 - Debug Probe Selection dialog box 38, 346
 - Debug (Report Assert option) 75
 - debug_file (cspybat option) 342
 - debugger concepts, definitions of 31
 - debugger drivers
 - simulator 36
 - debugger drivers. *See* C-SPY drivers
 - Debugger Macros window 329
 - debugger system overview 33
 - debugging projects
 - externally built applications 44
 - loading multiple images 45
 - debugging, RTOS awareness 31
 - Default breakpoint type (GDB Server option) 382
 - Default breakpoint type (I-jet option) 388
 - __delay (C-SPY system macro) 288
 - Delay (Autostep Settings option) 76
 - Delete/revert All Custom SFRs
(SFR Setup window context menu) 159
 - Device description file (debugger option) 375
 - device description files 43
 - definition of 47
 - modifying 47
 - specifying interrupts 302
 - device_macro (C-SPY command line option) 342
 - Disable Debugger Cache (I-jet menu) 396
 - __disableInterrupts (C-SPY system macro) 289
 - disable_interrupts (C-SPY command line option) 342
 - disable_misaligned_exception (C-SPY command line option) 343
 - Disassembly window 65
 - context menu 67, 240
 - disclaimer 2
 - DLIB
 - consuming breakpoints 109
 - naming convention 26

do (macro statement)	277
document conventions	24
documentation	
overview of guides.	23
overview of this guide	22
this guide	21
--download_only (C-SPY command line option)	343
Driver options (debugger option)	389
Driver (debugger option)	374
__driverType (C-SPY system macro)	289
--drv_catch_exceptions (C-SPY command line option)	344
--drv_communication (C-SPY command line option)	345
--drv_communication_log (C-SPY command line option)	346
--drv_default_breakpoint (C-SPY command line option)	346
--drv_exclude_from_verify (C-SPY command line option)	347
--drv_interface (C-SPY command line option)	347
--drv_interface_speed (C-SPY command line option)	348
--drv_reset_to_cpu_start (C-SPY command line option)	348
--drv_restore_breakpoints (C-SPY command line option)	348
--drv_suppress_download (C-SPY command line option)	337
--drv_system_bus_access (C-SPY command line option)	349
--drv_vector_table_base (C-SPY command line option)	350
--drv_verify_download (C-SPY command line option)	337
Duration (I-jet option).	384

E

Edit Breakpoint.	69
Edit Memory Range dialog box	160
Edit Memory Range dialog box (C-SPY simulator)	164
Edit Memory Range dialog box (C-SPY hardware debugger drivers)	169
Edit Nickname (Debug Probe Selection dialog box)	38
Edit Settings (Trace toolbar)	184
edition, of this guide	2
ELF entry point	
disabling warning when 0x0	363
Enable interrupt simulation (Interrupt Configuration option)	253

__enableInterrupts (C-SPY system macro)	290
Enable/Disable Breakpoint (Disassembly window context menu)	69
Enable/Disable (Trace toolbar)	183
End address (Memory Save option)	142
endianness. <i>See</i> byte order	
Enter Location dialog box.	126
ETB trace	176
ETM trace	176
__evaluate (C-SPY system macro)	290
Evaluate Now (Macro Quicklaunch window context menu)	332
examples	
C-SPY macros	269
interrupts	
interrupt logging	250
timer	248
macros	
checking status of register.	271
creating a log macro	272
using Quick Watch	271
performing tasks and continue execution	114
tracing incorrect function arguments	113
execUserAttach (C-SPY setup macro)	280
execUserCoreConnect (C-SPY setup macro)	283
execUserExecutionStarted (C-SPY setup macro)	280
execUserExecutionStopped (C-SPY setup macro)	281
execUserExit (C-SPY setup macro)	282
execUserFlashExit (C-SPY setup macro)	283
execUserFlashInit (C-SPY setup macro)	281
execUserFlashReset (C-SPY setup macro)	282
execUserPreload (C-SPY setup macro)	280
execUserPreReset (C-SPY setup macro)	282
execUserReset (C-SPY setup macro)	282
execUserSetup (C-SPY setup macro)	281
executed code, covering	227
execution history, tracing	179
Execution state (Cores window)	239
Explicit probe configuration (I-jet option)	387
expressions. <i>See</i> C-SPY expressions	

extended command line file, for cspybat 350
 Extra Options, for C-SPY 379

F

-f (cspybat option). 350
 Factory ranges (Memory Configuration option) 167
 File format (Memory Save option) 142
 file types
 device description, specifying in IDE 43
 macro 42, 375
 filename extensions
 ddf, selecting device description file 43
 mac, using macro file 42
 Filename (Memory Restore option) 143
 Filename (Memory Save option) 143
 Fill dialog box. 144
 __fillMemory8 (C-SPY system macro) 291
 __fillMemory16 (C-SPY system macro) 291
 __fillMemory32 (C-SPY system macro) 292
 __fillMemory64 (C-SPY system macro) 293
 Find in Trace dialog box 194
 Find in Trace window 195
 Find in Trace (Disassembly window context menu) 69
 Find (Memory window context menu) 141
 Find (Trace toolbar) 184
 first activation time (interrupt property), definition of . . . 244
 First activation (Interrupt Configuration option) 252
 flash loader
 parameters to control 369
 specifying relocation 369
 specifying the path to 369
 using 365
 Flash Loader Overview dialog box 366
 flash memory, load library module to 298
 --flash_loader (C-SPY command line option) 351
 __fmessage (C-SPY macro keyword) 277
 Focus on Core (Cores window context menu) 240
 for (macro statement) 277

Force Interrupt
 (Available Interrupts window context menu) 255
 Forced Interrupts (Simulator menu) 393
 Format
 (Registers User Groups Setup window context menu) . . . 156
 Function Profiler window 220
 Function Profiler (Simulator menu) 392
 function profiling
 real-time. 216
 Function Trace window 190
 functions
 C-SPY running to when starting 42, 374
 most time spent in, locating 215
 --function_profiling (cspybat option) 351

G

GDB Server
 breakpoint options 382
 setup options 381
 GDB Server menu (C-SPY driver) 396
 __gdbserver_exec_command (C-SPY system macro) . . . 294
 --gdbserv_exec_command (C-SPY command line option) 352
 __getArg (C-SPY system macro) 284
 __getNumberOfCores (C-SPY system macro) 295
 __getSelectedCore (C-SPY system macro) 295
 Go To Source
 (Timeline window context menu) 210, 265
 Go (Debug menu) 61

H

highlighting, in C-SPY 62
 Hold time (Interrupt Configuration option) 252
 hold time (interrupt property), definition of 245

I

IAR debugger driver plugin (debugger option) 389

icons, in this guide	25
if else (macro statement)	276
if (macro statement)	276
Ignore (Report Assert option)	75
Images window	53
Images, loading multiple	376
immediate breakpoints, overview	107
Input Mode dialog box	72
input, special characters in Terminal I/O window	72
installation directory	25
Instruction Profiling (Disassembly window context menu)	68
Intel-extended, C-SPY output format	34
Interface speed (I-jet option)	386
Interface (I-jet option)	386
Interrupt Configuration window	251
Interrupt Log graph in Timeline window	263
Interrupt Log Summary window	260
Interrupt Log Summary (Simulator menu)	392
Interrupt Log window	257
Interrupt Status window	255
interrupt system, using device description file	247
interrupts	
adapting C-SPY system for target hardware	247
simulated, introduction to	243
timer, example	248
using system macros	246
Interrupts (Timeline window context menu)	265
<code>__isBatchMode</code> (C-SPY system macro)	296
<code>__isMacroSymbolDefined</code> (C-SPY system macro)	296
italic style, in this guide	25
I-jet	
breakpoint options	388
hardware setup	36
interface options	386
setup options	383
solving communication problems	387
I-jet menu (C-SPY driver)	396
I/O register. <i>See</i> SFR	

J

<code>--jet_board_cfg</code> (C-SPY command line option)	352
<code>--jet_board_did</code> (C-SPY command line option)	353
<code>--jet_ir_length</code> (C-SPY command line option)	353
<code>--jet_itc_output</code> (C-SPY command line option)	354
<code>--jet_power_from_probe</code> (C-SPY command line option)	354
<code>--jet_script_file</code> (C-SPY command line option)	355
<code>--jet_sigprobe_opt</code> (C-SPY command line option)	355
<code>--jet_standard_reset</code> (C-SPY command line option)	357
<code>--jet_startup_connection_timeout</code> (C-SPY command line option)	358
<code>--jet_tap_position</code> (C-SPY command line option)	358
JTAG scan chain contains non-RISC-V devices (I-jet setting)	387

L

labels (assembler), viewing	82
Leave Target Running (I-jet menu)	397
<code>--leave_target_running</code> (C-SPY command line option)	359
Length (Fill option)	144
library functions	
C-SPY support for using, plugin module	362
lightbulb icon, in this guide	25
linker options	
typographic convention	25
consuming breakpoints	109
Little Endian (Memory window context menu)	140
Live Watch window	91
<code>__loadImage</code> (C-SPY system macro)	297
loading multiple debug files, list currently loaded	53
loading multiple images	45
Locals window	86
log breakpoints, overview	106
loop statements, in C-SPY macros	277

M

mac (filename extension), using a macro file 42
 --macro (C-SPY command line option) 359
 macro files, specifying 42, 375
 Macro Quicklaunch window 331
 Macro Registration window 327
 macro statements 276
 macros
 executing 269
 using 267
 --macro-param (C-SPY command line option) 360
 main function, C-SPY running to when starting 42, 374
 __makeString (C-SPY system macro) 284
 --mapu (C-SPY command line option) 360
 master project (multicore debugging) 234
 Memory access checking (Memory Access Setup option) 163
 Memory Configuration dialog box 166
 Memory Configuration dialog box (C-SPY simulator) . . 162
 Memory Configuration (GDB Server menu) 396
 Memory Fill (Memory window context menu) 141
 Memory Restore dialog box 143
 Memory Restore (Memory window context menu) 141
 Memory Save dialog box 142
 Memory Save (Memory window context menu) 141
 Memory window 138
 memory zones 130
 __memoryRestore (C-SPY system macro) 298
 __memorySave (C-SPY system macro) 299
 menu bar, C-SPY-specific 49
 __message (C-SPY macro keyword) 277
 __messageBoxYesCancel (C-SPY system macro) 300
 __messageBoxYesNo (C-SPY system macro) 301
 Messages window, amount of output 74
 misaligned data, detecting 394
 Mixed Mode (Disassembly window context menu) 69
 monospace font, meaning of in guide. *See* computer style
 Motorola, C-SPY output format 34
 Move to PC (Disassembly window context menu) 67

multicore debugging 233
 asymmetric multicore debugging 234
 session file 241
 Multicore toolbar 241
 Multicore (C-SPY options) 377
 --multicore_nr_of_cores (C-SPY command line option) . 361
 Multi-target debug system (I-jet setting) 387

N

Name (Edit SFR option) 160
 naming conventions 26
 Navigate
 (Timeline window context menu) 209, 264
 Next Symbol (Symbolic Memory window context menu) 147
 Notification (Data Alignment Setup option) 394
 Number of cores (debugger option) 377

O

__openFile (C-SPY system macro) 301
 Operation (Fill option) 144
 operators, sizeof in C-SPY 80
 optimizations, effects on variables 80
 options
 in the IDE 373
 on the command line 338, 379
 Options (Stack window context menu) 151
 __orderInterrupt (C-SPY system macro) 302
 Originator (debugger option) 380
 Override default delay after
 reset (I-jet option) 384
 Override default .board file (debugger option) 376

P

-p (C-SPY command line option) 361
 __param (C-SPY macro keyword) 275
 parameters

list of passed to the flash loader	367
tracing incorrect values of	63
typographic convention	25
part number, of this guide	2
partner project (multicore debugging).	234
PC (Cores window).	240
Performance Monitoring (I-jet menu)	398
peripheral units	
device-specific	47
displayed in Registers window.	130
in C-SPY expressions	79
initializing using setup macros.	268
peripherals register. <i>See</i> SFR	
Please select one symbol	
(Resolve Symbol Ambiguity option)	103
--plugin (C-SPY command line option)	361
plugin modules (C-SPY).	34
loading.	43
Plugins (C-SPY options).	380
__popSimulatorInterruptExecutingStack	
(C-SPY system macro)	303
pop-up menu. <i>See</i> context menu	
Preceding bits (I-jet setting)	387
prerequisites, programming experience	21
Previous Symbol	
(Symbolic Memory window context menu)	147
probability (interrupt property)	252
definition of	244
Probability % (Interrupt Configuration option).	252
Probe config (I-jet option)	386
Probe configuration file (I-jet option)	387
__probeType (C-SPY system macro)	304
Profile Selection (Timeline window context menu)	212
profiling	
analyzing data	217
on function level	217
on instruction level.	219
profiling information, on functions and instructions	215
profiling sources	
trace (calls)	216

trace (flat)	216
program execution	
breaking	106–107
in C-SPY	57
multiple cores in C-SPY	233
programming experience	21
program. <i>See</i> application	
Progress bar (Trace toolbar)	184
projects, for debugging externally built applications.	44
publication date, of this guide.	2

Q

Quick Watch window	97
executing C-SPY macros	271

R

RAM (Edit Memory Access option).	170
Range for (Viewing Range option)	212
__readFile (C-SPY system macro)	304
__readFileByte (C-SPY system macro)	305
reading guidelines	21
__readMemoryBuffer (C-SPY system macro)	285
__readMemoryByte (C-SPY system macro).	306
__readMemory8 (C-SPY system macro)	306
__readMemory16 (C-SPY system macro)	306
__readMemory32 (C-SPY system macro)	307
__readMemory64 (C-SPY system macro)	307
reference information, typographic convention.	25
register groups	130
predefined, enabling.	152
Register User Groups Setup window	155
registered trademarks	2
__registerMacroFile (C-SPY system macro)	308
Registers window	152
registers, displayed in Registers window	152
Removal All Groups	
(Registers User Groups Setup window context menu)	156

- Removal
 - (Registers User Groups Setup window context menu) . . . 156
 - Remove All (Macro Quicklaunch window context menu) 332
 - Remove (Macro Quicklaunch window context menu) . . . 332
 - Repeat interval (Interrupt Configuration option) 252
 - repeat interval (interrupt property), definition of 244
 - Replace (Memory window context menu) 141
 - Report Assert dialog box 75
 - Reset (I-jet option) 383
 - __resetFile (C-SPY system macro) 308
 - reset_style (C-SPY command line option) 362
 - Resolve Source Ambiguity dialog box 128
 - Restore software breakpoints at (GDB Server option) . . . 382
 - Restore (Memory Restore option) 143
 - return (macro statement) 277
 - ROM/Flash (Edit Memory Access option) 170
 - RTOS awareness debugging 31
 - RTOS awareness (C-SPY plugin module) 31
 - Run to Cursor (Disassembly window context menu) 67
 - Run to Cursor, command for executing 62
 - Run to (C-SPY option) 42, 374
 - Run/Step/Stop affect all cores
 - (Cores window context menu) 240
 - Run/Step/Stop affect current core only
 - (Cores window context menu) 240
- S**
 - Save Custom SFRs (SFR Setup window context menu) . . 160
 - Save to File (Timeline window context menu) 210
 - Save to File
 - (Register User Groups Setup window context menu) . . . 157
 - Save (Memory Save option) 143
 - Save (Trace toolbar) 184
 - Scale (Viewing Range option) 213
 - scripting C-SPY. *See* macros
 - Select Graphs
 - (Timeline window context menu) 211, 265
 - Select plugins to load (debugger option) 380
 - __selectCore (C-SPY system macro) 308
 - Serial trace 176
 - session file (multicore debugging) 241
 - Set Data Breakpoint (Memory window context menu) . . . 141
 - Set Data Log Breakpoint
 - (Memory window context menu) 142
 - Set Next Statement (Disassembly window context menu) . 69
 - __setCodeBreak (C-SPY system macro) 309
 - __setDataBreak (C-SPY system macro) 310
 - __setDataLogBreak (C-SPY system macro) 312
 - __setLogBreak (C-SPY system macro) 313
 - __setSimBreak (C-SPY system macro) 314
 - __setTraceStartBreak (C-SPY system macro) 315
 - __setTraceStopBreak (C-SPY system macro) 316
 - setup macro file, registering 42
 - setup macro functions 268
 - reserved names. 279
 - Setup macros (debugger option) 375
 - Setup (C-SPY options) 374
 - SFR
 - in Registers window. 153
 - using as assembler symbols 79
 - SFR Setup window 157
 - SFR/Uncached (Edit Memory Access option) 170
 - shortcut menu. *See* context menu
 - Show All (SFR Setup window context menu) 159
 - Show Custom SFRs only
 - (SFR Setup window context menu) 159
 - Show Factory SFRs only
 - (SFR Setup window context menu) 159
 - Show offsets (Stack window context menu) 150
 - Show Timing (Timeline window context menu) 210
 - Show variables (Stack window context menu) 150
 - silent (C-SPY command line option) 363
 - Simple (multicore debugger option) 377
 - Simulated Frequency dialog box 395
 - simulating interrupts, enabling/disabling 253
 - Simulator menu. 392
 - simulator, introduction 36
 - Size (Edit SFR option) 161
 - sizeof 80

slave. <i>See</i> partner	
__smessage (C-SPY macro keyword)	277
Sort by (Timeline window context menu)	265
__sourcePosition (C-SPY system macro)	317
special function registers (SFR)	
in Registers window	153
using as assembler symbols	79
Stack window	148
standard C, sizeof operator in C-SPY	80
Start address (Fill option)	144
Start address (Memory Save option)	142
Start Core (Cores window context menu)	240
static analysis tool, documentation for	24
Statics window	94
Status (Cores window)	240
Step Into, description	59
Step Out, description	60
Step Over, description	59
step points, definition of	58
Stop Core (Cores window context menu)	240
__strFind (C-SPY system macro)	317
__subString (C-SPY system macro)	318
Suppress download (debugger option)	375
--suppress_download (C-SPY command line option)	349
--suppress_entrypoint_warning (C-SPY command line option)	363
Symbolic Memory window	145
Symbols window	100
symbols, in C-SPY expressions	78
__system1 (C-SPY system macro)	319
__system2 (C-SPY system macro)	319
__system3 (C-SPY system macro)	320

T

Target number (I-jet setting)	387
Target power (I-jet option)	384
target system, definition of	33
Target with multiple CPUs (I-jet setting)	387

__targetDebuggerVersion (C-SPY system macro)	321
TCP/IP address or hostname (GDB Server option)	381
Terminal IO Log Files (Terminal IO Log Files option)	73
Terminal I/O Log Files dialog box	73
Terminal I/O window	63, 72
Text search (Find in Trace option)	194
Third-Party Driver (debugger options)	389
Time Axis Unit	
(Timeline window context menu)	211, 265
Timeline window	263
Timeline window (Call Stack graph)	207
--timeout (C-SPY command line option)	364
timer interrupt, example	248
Toggle Breakpoint (Code)	
(Disassembly window context menu)	68
Toggle Breakpoint (Log)	
(Disassembly window context menu)	68
Toggle Breakpoint (Trace Start)	
(Disassembly window context menu)	68
Toggle Breakpoint (Trace Stop)	
(Disassembly window context menu)	68
Toggle source (Trace toolbar)	184
__toLower (C-SPY system macro)	321
tools icon, in this guide	25
__toString (C-SPY system macro)	322
__toUpper (C-SPY system macro)	322
trace	175, 197
trace mode	
setting from the command line	355
setting in the IDE	180
Trace Settings button (IDE toolbar)	179
Trace Settings dialog box	180
Trace Start Trigger breakpoint dialog box	191
trace start/stop trigger breakpoints, overview	106
Trace Stop Trigger breakpoint dialog box	193
Trace window	183
trace (calls), profiling source	216
trace (flat), profiling source	216
trademarks	2
typographic conventions	25

U

- Unavailable, C-SPY message 81
- __unloadImage (C-SPY system macro) 323
- Use command line options (debugger option). 379
- Use Extra Images (debugger option). 376
- Use flash loader (debugger option) 376
- Use manual ranges (Memory Access Setup option) 163
- Use ranges based on (Memory Access Setup option) 162
- Used ranges (Memory Configuration option) 167
- user application, definition of 33

V

- Value (Fill option) 144
- __var (C-SPY macro keyword). 274
- variables
 - effects of optimizations 80
 - in C-SPY expressions 78
 - information, limitation on 80
- variance (interrupt property), definition of 245
- Variance % (Interrupt Configuration option) 252
- Verify download (debugger option) 375
- verify_download (C-SPY command line option) 350
- version
 - of this guide 2
- Viewing Range dialog box 212
- Visual State, C-SPY plugin module for 34

W

- __wallTime_ms (C-SPY system macro). 323
- warnings icon, in this guide 26
- Watch window 88
 - using 77
- web sites, recommended 24
- while (macro statement) 277
- windows, specific to C-SPY 52
- __writeFile (C-SPY system macro) 324

- __writeFileByte (C-SPY system macro). 324
- __writeMemoryBuffer (C-SPY system macro). 286
- __writeMemoryByte (C-SPY system macro) 325
- __writeMemory8 (C-SPY system macro). 325
- __writeMemory16 (C-SPY system macro). 325
- __writeMemory32 (C-SPY system macro). 326
- __writeMemory64 (C-SPY system macro). 326

Z

- zone
 - in C-SPY 130
 - part of an absolute address. 127
- Zone (Edit SFR option). 161
- Zoom
 - (Timeline window context menu) 209, 264

Symbols

- __abortLaunch (C-SPY system macro). 286
- __argCount (C-SPY system macro) 283
- __bytes2Word16 (C-SPY system macro) 283
- __bytes2Word32 (C-SPY system macro) 283
- __cancelAllInterrupts (C-SPY system macro) 287
- __cancelInterrupt (C-SPY system macro). 287
- __clearBreak (C-SPY system macro) 287
- __closeFile (C-SPY system macro) 288
- __delay (C-SPY system macro) 288
- __disableInterrupts (C-SPY system macro) 289
- __driverType (C-SPY system macro). 289
- __enableInterrupts (C-SPY system macro). 290
- __evaluate (C-SPY system macro) 290
- __fillMemory8 (C-SPY system macro) 291
- __fillMemory16 (C-SPY system macro). 291
- __fillMemory32 (C-SPY system macro). 292
- __fillMemory64 (C-SPY system macro). 293
- __fmessage (C-SPY macro keyword). 277
- __gdbserver_exec_command (C-SPY system macro). . . . 294
- __getArg (C-SPY system macro) 284

__getNumberOfCores (C-SPY system macro)	295	__system2 (C-SPY system macro)	319
__getSelectedCore (C-SPY system macro)	295	__system3 (C-SPY system macro)	320
__isBatchMode (C-SPY system macro)	296	__targetDebuggerVersion (C-SPY system macro)	321
__isMacroSymbolDefined (C-SPY system macro)	296	__toLower (C-SPY system macro)	321
__loadImage (C-SPY system macro)	297	__toString (C-SPY system macro)	322
__makeString (C-SPY system macro)	284	__ToUpper (C-SPY system macro)	322
__memoryRestore (C-SPY system macro)	298	__unloadImage (C-SPY system macro)	323
__memorySave (C-SPY system macro)	299	__var (C-SPY macro keyword)	274
__message (C-SPY macro keyword)	277	__wallTime_ms (C-SPY system macro)	323
__messageBoxYesCancel (C-SPY system macro)	300	__writeFile (C-SPY system macro)	324
__messageBoxYesNo (C-SPY system macro)	301	__writeFileByte (C-SPY system macro)	324
__openFile (C-SPY system macro)	301	__writeMemoryBuffer (C-SPY system macro)	286
__orderInterrupt (C-SPY system macro)	302	__writeMemoryByte (C-SPY system macro)	325
__param (C-SPY macro keyword)	275	__writeMemory8 (C-SPY system macro)	325
__popSimulatorInterruptExecutingStack (C-SPY system macro)	303	__writeMemory16 (C-SPY system macro)	325
__probeType (C-SPY system macro)	304	__writeMemory32 (C-SPY system macro)	326
__readFile (C-SPY system macro)	304	__writeMemory64 (C-SPY system macro)	326
__readFileByte (C-SPY system macro)	305	-f (cspybat option)	350
__readMemoryBuffer (C-SPY system macro)	285	-p (C-SPY command line option)	361
__readMemoryByte (C-SPY system macro)	306	--application_args (C-SPY command line option)	338
__readMemory8 (C-SPY system macro)	306	--attach_to_running_target (C-SPY command line option)	339
__readMemory16 (C-SPY system macro)	306	--backend (C-SPY command line option)	340
__readMemory32 (C-SPY system macro)	307	--code_coverage_file (C-SPY command line option)	340
__readMemory64 (C-SPY system macro)	307	--core (C-SPY command line option)	341
__registerMacroFile (C-SPY system macro)	308	--cycles (C-SPY command line option)	341
__resetFile (C-SPY system macro)	308	--debug_file (cspybat option)	342
__selectCore (C-SPY system macro)	308	--device_macro (C-SPY command line option)	342
__setCodeBreak (C-SPY system macro)	309	--disable_interrupts (C-SPY command line option)	342
__setDataBreak (C-SPY system macro)	310	--disable_misaligned_exception (C-SPY command line option)	343
__setDataLogBreak (C-SPY system macro)	312	--download_only (C-SPY command line option)	343
__setLogBreak (C-SPY system macro)	313	--drv_catch_exceptions (C-SPY command line option)	344
__setSimBreak (C-SPY system macro)	314	--drv_communication (C-SPY command line option)	345
__setTraceStartBreak (C-SPY system macro)	315	--drv_communication_log (C-SPY command line option)	346
__setTraceStopBreak (C-SPY system macro)	316	--drv_default_breakpoint (C-SPY command line option)	346
__smessage (C-SPY macro keyword)	277	--drv_exclude_from_verify (C-SPY command line option)	347
__sourcePosition (C-SPY system macro)	317	--drv_interface (C-SPY command line option)	347
__strFind (C-SPY system macro)	317		
__subString (C-SPY system macro)	318		
__system1 (C-SPY system macro)	319		

--drv_interface_speed (C-SPY command line option) . . . 348
 --drv_reset_to_cpu_start (C-SPY command line option) . 348
 --drv_restore_breakpoints
 (C-SPY command line option) 348
 --drv_suppress_download
 (C-SPY command line option) 337
 --drv_system_bus_access (C-SPY command line option). 349
 --drv_vector_table_base (C-SPY command line option). . 350
 --drv_verify_download (C-SPY command line option) . . 337
 --flash_loader (C-SPY command line option). 351
 --function_profiling (cspybat option) 351
 --gdbserve_exec_command
 (C-SPY command line option) 352
 --jet_board_cfg (C-SPY command line option) 352
 --jet_board_did (C-SPY command line option) 353
 --jet_ir_length (C-SPY command line option) 353
 --jet_itc_output (C-SPY command line option) 354
 --jet_power_from_probe (C-SPY command line option) . 354
 --jet_script_file (C-SPY command line option) 355
 --jet_sigprobe_opt (C-SPY command line option) 355
 --jet_standard_reset (C-SPY command line option) 357
 --jet_startup_connection_timeout
 (C-SPY command line option) 358
 --jet_tap_position (C-SPY command line option). 358
 --leave_target_running (C-SPY command line option). . . 359
 --macro (C-SPY command line option) 359
 --macro_param (C-SPY command line option). 360
 --mapu (C-SPY command line option) 360
 --multicore_nr_of_cores (C-SPY command line option). . 361
 --plugin (C-SPY command line option) 361
 --reset_style (C-SPY command line option). 362
 --silent (C-SPY command line option) 363
 --suppress_download (C-SPY command line option) . . . 349
 --suppress_entrypoint_warning (C-SPY command line option)
 363
 --timeout (C-SPY command line option) 364
 --verify_download (C-SPY command line option) 350

Numerics

1x Units (Symbolic Memory window context menu) 147
 8x Units (Memory window context menu) 140