



**IAR Embedded
Workbench**

C-SPY® Debugging Guide

for the Renesas
RX Family

COPYRIGHT NOTICE

© 2009–2023 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.

Renesas is a registered trademark of Renesas Electronics Corporation. RX is a trademark of Renesas Electronics Corporation.

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

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

EDITION NOTICE

Tenth edition: August 2023

Part number: UCSRX-10

This guide applies to version 5.x of IAR Embedded Workbench® for the Renesas RX family.

The *C-SPY® Debugging Guide for RX* replaces all debugging information in the *IAR Embedded Workbench IDE User Guide*. It also replaces the *C-SPY® Power Debugging Guide for RX* and the *IAR C-SPY® Hardware Debugger Systems User Guide for RX*.

Internal reference: FF9.2, ISHP.

Brief contents

Tables	21
Preface	23
Part 1. Basic debugging	31
The IAR C-SPY Debugger	33
Getting started using C-SPY	45
Executing your application	71
Variables and expressions	97
Breakpoints	123
Memory and registers	151
Part 2. Analyzing your application	187
Trace	189
The application timeline	211
Profiling	241
Analyzing code performance	255
Code coverage	263
Power debugging	269
C-RUN runtime error checking	289
Part 3. Advanced debugging	329
Interrupts	331
C-SPY macros	355
The C-SPY command line utility—cspybat	415

Part 4. Additional reference information	437
Debugger options	439
Additional information on C-SPY drivers	449
Index	463

Contents

Tables	21
Preface	23
Who should read this guide	23
Required knowledge	23
How to use this guide	23
What this guide contains	24
Part 1. Basic debugging	24
Part 2. Analyzing your application	24
Part 3. Advanced debugging	25
Part 4. Additional reference information	25
Other documentation	25
User and reference guides	26
The online help system	26
Web sites	27
Document conventions	27
Typographic conventions	27
Naming conventions	28
 Part I. Basic debugging	 31
The IAR C-SPY Debugger	33
Introduction to C-SPY	33
An integrated environment	33
General C-SPY debugger features	34
RTOS awareness	35
Debugger concepts	36
C-SPY and target systems	36
The debugger	37
The target system	37
The application	37
C-SPY debugger systems	38

The ROM-monitor program	38
Third-party debuggers	38
C-SPY plugin modules	38
C-SPY drivers overview	39
Differences between the C-SPY drivers	39
The IAR C-SPY Simulator	40
Supported features	40
The C-SPY EI/E20 and E2/E2 Lite/EZ-CUBE2 drivers	41
Communication overview	41
Hardware installation	42
The C-SPY J-Link driver	42
Communication overview	42
Hardware installation	43
Getting started using C-SPY	45
Setting up C-SPY	45
Setting up for debugging	45
Executing from reset	46
Using a setup macro file	46
Selecting a device description file	47
Loading plugin modules	47
Starting C-SPY	47
Starting a debug session	48
Loading executable files built outside of the IDE	48
Starting a debug session with source files missing	48
Loading multiple debug images	49
Editing in C-SPY windows	50
Downloading files to external flash memory	51
Start debugging a running application	52
Adapting for target hardware	53
Modifying a device description file	53
Initializing target hardware before C-SPY starts	54
Reference information on starting C-SPY	55
C-SPY Debugger main window	55

Images window	59
Get Alternative File dialog box	60
Download Emulator Firmware dialog box	61
Operating Frequency dialog box	62
Hardware Setup dialog box: MCU	63
Hardware Setup dialog box: External Memory	66
External Area dialog box	68
Executing your application	71
Introduction to application execution	71
Briefly about application execution	71
Source and disassembly mode debugging	71
Single stepping	72
Troubleshooting slow stepping speed	74
Running the application	75
Highlighting	76
Viewing the call stack	77
Terminal input and output	77
Debug logging	78
Reference information on application execution	78
Disassembly window	79
Call Stack window	84
Terminal I/O window	86
Terminal I/O Log File dialog box	87
Debug Log window	88
Report Assert dialog box	89
Start/Stop Function Settings dialog box	90
Select Label dialog box	92
Autostep settings dialog box	93
ID Code Verification dialog box	93
Cores window	94
Variables and expressions	97
Introduction to working with variables and expressions	97
Briefly about working with variables and expressions	97

C-SPY expressions	98
Limitations on variable information	100
Working with variables and expressions	101
Using the windows related to variables and expressions	101
Viewing assembler variables	102
Reference information on working with variables and expressions	103
Auto window	103
Locals window	106
Watch window	108
Live Watch window	111
Statics window	113
Quick Watch window	116
Symbols window	119
Resolve Symbol Ambiguity dialog box	122
Breakpoints	123
Introduction to setting and using breakpoints	123
Reasons for using breakpoints	123
Briefly about setting breakpoints	123
Breakpoint types	124
Breakpoint icons	126
Breakpoints in the C-SPY simulator	127
Breakpoints in the C-SPY hardware debugger drivers	127
Breakpoint consumers	127
Setting breakpoints	128
Various ways to set a breakpoint	129
Toggling a simple code breakpoint	129
Setting breakpoints using the dialog box	129
Setting a data breakpoint in the Memory window	131
Setting breakpoints using system macros	131
Useful breakpoint hints	132
Reference information on breakpoints	133
Breakpoints window	134

Breakpoint Usage window	136
Code breakpoints dialog box	137
Hardware Code Breakpoint dialog box	138
Software Code Breakpoint dialog box	140
Log breakpoints dialog box	141
Data breakpoints dialog box (Simulator)	142
Data breakpoints dialog box (C-SPY hardware debugger drivers) ..	144
Data Log breakpoints dialog box	146
Immediate breakpoints dialog box	147
Enter Location dialog box	148
Resolve Source Ambiguity dialog box	150
Memory and registers	151
Introduction to monitoring memory and registers	151
Briefly about monitoring memory and registers	151
C-SPY memory zones	153
Memory configuration for the C-SPY simulator	153
Monitoring memory and registers	154
Defining application-specific register groups	154
Monitoring stack usage	155
Reference information on memory and registers	158
Memory window	159
Memory Save dialog box	163
Memory Restore dialog box	164
Fill dialog box	165
Symbolic Memory window	166
Stack window	169
Registers window	173
Register User Groups Setup window	176
SFR Setup window	178
Edit SFR dialog box	181
Memory Access Setup dialog box	183
Edit Memory Access dialog box	185

Part 2. Analyzing your application	187
Trace	189
Introduction to using trace	189
Reasons for using trace	189
Briefly about trace	189
Requirements for using trace	190
Collecting and using trace data	190
Getting started with trace	190
Trace data collection using breakpoints	191
Searching in trace data	191
Browsing through trace data	192
Reference information on trace	192
Trace Settings dialog box	193
Trace window	196
Function Trace window	203
Trace Start Trigger breakpoint dialog box	205
Trace Stop Trigger breakpoint dialog box	206
Data Trace Collection breakpoints dialog box	207
Find in Trace dialog box	207
Find in Trace window	209
The application timeline	211
Introduction to analyzing your application's timeline	211
Briefly about analyzing the timeline	211
Requirements for timeline support	213
Analyzing your application's timeline	213
Displaying a graph in the Timeline window	213
Navigating in the graphs	214
Analyzing performance using the graph data	214
Getting started using data logging	215
Getting started using data sampling	216
Reference information on application timeline	217
Timeline window—Call Stack graph	218

Timeline window—Data Log graph	221
Data Log window	225
Data Log Summary window	228
Data Sample window	230
Data Sample Setup window	232
Sampled Graphs window	234
Viewing Range dialog box	238
Profiling	241
Introduction to the profiler	241
Reasons for using the profiler	241
Briefly about the profiler	241
Requirements for using the profiler	242
Using the profiler	243
Getting started using the profiler on function level	243
Analyzing the profiling data	244
Getting started using the profiler on instruction level	246
Selecting a time interval for profiling information	247
Reference information on the profiler	248
Function Profiler window	249
Analyzing code performance	255
Introduction to performance analysis	255
Reasons for using performance analysis	255
Briefly about performance analysis	255
Requirements for performance analysis	256
Analyzing performance	256
Using performance analysis	256
Reference information on performance analysis	256
Performance Analysis Setup dialog box	257
Performance Analysis window	259
Performance Start breakpoints dialog box	261
Performance Stop breakpoints dialog box	262

Code coverage	263
Introduction to code coverage	263
Reasons for using code coverage	263
Briefly about code coverage	263
Requirements and restrictions for using code coverage	263
Using code coverage	264
Getting started using code coverage	264
Reference information on code coverage	264
Code Coverage window	265
Power debugging	269
Introduction to power debugging	269
Reasons for using power debugging	269
Briefly about power debugging	269
Requirements and restrictions for power debugging	270
Optimizing your source code for power consumption	271
Waiting for device status	271
Software delays	271
DMA versus polled I/O	272
Low-power mode diagnostics	272
CPU frequency	273
Detecting mistakenly unattended peripherals	273
Peripheral units in an event-driven system	273
Finding conflicting hardware setups	274
Analog interference	275
Debugging in the power domain	275
Displaying a power profile and analyzing the result	276
Detecting unexpected power usage during application execution ...	276
Changing the graph resolution	277
Reference information on power debugging	277
Power Log Setup window	278
Power Log window	281
Timeline window—Power graph	285

C-RUN runtime error checking	289
Introduction to runtime error checking	289
Runtime error checking	289
Runtime error checking using C-RUN	290
The checked heap provided by the library	291
Using C-RUN in the IAR Embedded Workbench IDE	291
Using C-RUN in non-interactive mode	292
Requirements for runtime error checking	292
Using C-RUN	292
Getting started using C-RUN runtime error checking	293
Creating rules for messages	295
Detecting various runtime errors	295
Detecting implicit or explicit integer conversion	295
Detecting signed or unsigned overflow	297
Detecting bit loss or undefined behavior when shifting	299
Detecting division by zero	300
Detecting unhandled cases in switch statements	300
Detecting accesses outside the bounds of arrays and other objects	301
Detecting heap usage error	308
Detecting heap memory leaks	309
Detecting heap integrity violations	311
Reference information on runtime error checking	314
C-RUN Runtime Checking options	314
C-RUN Messages window	316
C-RUN Messages Rules window	318
Compiler and linker reference for C-RUN	320
--bounds_table_size	321
--debug_heap	321
--generate_entries_without_bounds	321
--ignore_uninstrumented_pointers	322
--ignore_uninstrumented_pointers	322
--runtime_checking	322
#pragma default_no_bounds	323

#pragma define_with_bounds	324
#pragma define_without_bounds	324
#pragma disable_check	324
#pragma generate_entry_without_bounds	325
#pragma no_arith_checks	325
#pragma no_bounds	325
__as_get_base	326
__as_get_bound	326
__as_make_bounds	326
cspybat options for C-RUN	327
--rtc_enable	327
--rtc_output	327
--rtc_raw_to_txt	328
--rtc_rules	328

Part 3. Advanced debugging

Interrupts	331
Introduction to interrupts	331
Briefly about the interrupt simulation system	331
Interrupt characteristics	332
Interrupt simulation states	333
C-SPY system macros for interrupt simulation	334
Target-adapting the interrupt simulation system	335
Briefly about interrupt logging	335
Using the interrupt system	335
Simulating a simple interrupt	336
Simulating an interrupt in a multi-task system	337
Getting started using interrupt logging	338
Reference information on interrupts	338
Interrupt Setup dialog box	339
Edit Interrupt dialog box	341
Forced Interrupt window	342
Interrupt Status window	343

Interrupt Log window	345
Interrupt Log Summary window	348
Timeline window—Interrupt Log graph	350
C-SPY macros	355
Introduction to C-SPY macros	355
Reasons for using C-SPY macros	355
Briefly about using C-SPY macros	356
Briefly about setup macro functions and files	356
Briefly about the macro language	356
Using C-SPY macros	357
Registering C-SPY macros—an overview	358
Executing C-SPY macros—an overview	358
Registering and executing using setup macros and setup files	359
Executing macros using Quick Watch	359
Executing a macro by connecting it to a breakpoint	360
Aborting a C-SPY macro	361
Reference information on the macro language	362
Macro functions	362
Macro variables	362
Macro parameters	363
Macro strings	363
Macro statements	364
Formatted output	365
Reference information on reserved setup macro function names	367
execUserAttach	367
execUserPreload	368
execUserSetup	368
execUserPreReset	368
execUserReset	369
execUserExit	369
Reference information on C-SPY system macros	369
__abortLaunch	372

__cancelAllInterrupts	372
__cancelInterrupt	372
__clearBreak	373
__closeFile	373
__delay	374
__disableInterrupts	374
__driverType	374
__enableInterrupts	375
__evaluate	375
__fillMemory8	376
__fillMemory16	377
__fillMemory32	378
__getNumberOfCores	379
__getSelectedCore	379
__isBatchMode	379
__isMacroSymbolDefined	380
__loadImage	380
__memoryRestore	382
__memorySave	382
__messageBoxYesCancel	383
__messageBoxYesNo	384
__openFile	385
__orderInterrupt	386
__popSimulatorInterruptExecutingStack	387
__readFile	387
__readFileByte	388
__readMemory8, __readMemoryByte	388
__readMemory16	389
__readMemory32	389
__registerMacroFile	390
__resetFile	390
__selectCore	391
__setCodeBreak	391
__setDataBreak	392

__setDataLogBreak	393
__setLogBreak	394
__setSimBreak	395
__setTraceStartBreak	396
__setTraceStopBreak	397
__sourcePosition	398
__strFind	398
__subString	399
__system1	400
__system2	400
__system3	401
__targetDebuggerVersion	402
__toLower	402
__toString	403
__toUpper	403
__unloadImage	404
__wallTime_ms	404
__writeFile	405
__writeFileByte	405
__writeMemory8, __writeMemoryByte	406
__writeMemory16	406
__writeMemory32	407
Graphical environment for macros	407
Macro Registration window	408
Debugger Macros window	410
Macro Quicklaunch window	412
The C-SPY command line utility—cspybat	415
Using C-SPY in batch mode	415
Before running cspybat for the first time	415
Starting cspybat	416
Output	416
Invocation syntax	417

Summary of C-SPY command line options	417
General cspybat options	418
Options available for all C-SPY drivers	419
Options available for the simulator driver	419
Options available for all C-SPY hardware debugger drivers	419
Options available for	
the E1/E20 and E2/E2 Lite/EZ-CUBE2 drivers	420
Options available for the J-Link driver	420
Reference information on C-SPY command line options	420
--application_args	420
--attach_to_running_target	421
--backend	421
--core	422
--code_coverage_file	422
--cspybat_inifile	423
--cycles	423
-d	423
--debug_file	424
--device_select	424
--diag_warning	425
--disable_interrupts	425
--double	425
--download_only	426
--drv_communication	426
--drv_mode	427
--endian	427
-f	428
--flash_only_changed_blocks	428
--fpu	429
--function_profiling	429
--int	430
--ir_length	430
--leave_target_running	431
--log_file	431

--macro	431
--macro_param	432
--mapu	432
-p	433
--plugin	433
--set_pc_to_entry_symbol	434
--silent	434
--suppress_download	434
--timeout	435
--verify_download	435
Part 4. Additional reference information	437
Debugger options	439
Setting debugger options	439
Reference information on general debugger options	440
Setup	440
Images	441
Plugins	442
Extra Options	443
Reference information on C-SPY hardware debugger driver options	444
Communication	444
Download	445
JTAG Scan Chain	446
Additional information on C-SPY drivers	449
Reference information on C-SPY driver menus	449
<i>C-SPY driver</i>	449
Simulator menu	450
E1/E20 Emulator menu	452
E2/E2 Lite/EZ-CUBE2 menu	454
J-Link menu	457

Reference information on the C-SPY simulator	459
Simulated Frequency dialog box	459
Reference information on the C-SPY hardware debugger	
drivers	459
Emulator information window	460
Resolving problems	460
Write failure during load	461
No contact with the target hardware	461
Index	463

Tables

1: Typographic conventions used in this guide	27
2: Naming conventions used in this guide	28
3: Driver differences	39
4: Restrictions on registers and flags	91
5: MCU status when the user application starts executing	91
6: C-SPY assembler symbols expressions	99
7: Handling name conflicts between hardware registers and assembler labels	99
8: Available breakpoints in C-SPY hardware debugger drivers	127
9: C-SPY macros for breakpoints	131
10: Supported graphs in the Timeline window	213
11: C-SPY driver profiling support	243
12: Project options for enabling the profiler	243
13: Project options for enabling code coverage	264
14: Timer interrupt settings	337
15: Examples of C-SPY macro variables	363
16: Summary of system macros	369
17: __cancelInterrupt return values	373
18: __disableInterrupts return values	374
19: __driverType return values	375
20: __enableInterrupts return values	375
21: __evaluate return values	376
22: __isBatchMode return values	379
23: __loadImage return values	381
24: __messageBoxYesCancel return values	384
25: __messageBoxYesNo return values	384
26: __openFile return values	385
27: __readFile return values	387
28: __setCodeBreak return values	391
29: __setDataBreak return values	392
30: __setDataLogBreak return values	393
31: __setLogBreak return values	394

32: __setSimBreak return values	396
33: __setTraceStartBreak return values	396
34: __setTraceStopBreak return values	397
35: __sourcePosition return values	398
36: __unloadImage return values	404
37: cspybat parameters	417
38: Options specific to the C-SPY drivers you are using	439

Preface

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

Who should read this guide

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

REQUIRED KNOWLEDGE

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

- The architecture and instruction set of the RX microcontroller family (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 25.

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 RX.

PART 1. 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.

- *Analyzing code performance* describes how to use a C-SPY hardware debugger to analyze code performance in terms of time, clock cycles, interrupts, exceptions, and instructions.
- *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.
- *Power debugging* describes techniques for power debugging and how you can use C-SPY to find source code constructions that result in unexpected power consumption.
- *C-RUN runtime error checking* describes how to use C-RUN for runtime error checking.

PART 3. ADVANCED DEBUGGING

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

PART 4. ADDITIONAL REFERENCE INFORMATION

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

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the 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 RX*.
- Using the IAR C-SPY® Debugger and C-RUN runtime error checking, is available in the *C-SPY® Debugging Guide for RX*.
- Programming for the IAR C/C++ Compiler for RX and linking, is available in the *IAR C/C++ Development Guide for RX*.
- Programming for the IAR Assembler for RX, is available in the *IAR Assembler Reference Guide for RX*.
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for RX, is available in the *IAR Embedded Workbench® Migration Guide*.
- Migrating from an older UBROF-based product version to a newer version that uses the ELF/DWARF object format, is available in the guide *IAR Embedded Workbench® Migrating from UBROF to ELF/DWARF*.
- Migrating from the Renesas High-performance Embedded Workshop and e2studio toolchains for RX to IAR Embedded Workbench® for RX, is available in the guide *Migrating from Renesas to IAR Embedded Workbench*.

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 Renesas web site, **www.renesas.com**, that contains information and news about the RX microcontrollers.
- 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 `rx\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Embedded Workbench N.n\rx\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 <i>filename.h</i> 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.

Table 1: Typographic conventions used in this guide





Style	Used for
{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.
	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 RX	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for RX	the IDE
IAR C-SPY® Debugger for RX	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for RX	the compiler
IAR Assembler™ for RX	the assembler
IAR ILINK Linker™	ILINK, the linker

Table 2: Naming conventions used in this guide

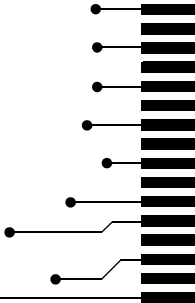
Brand name	Generic term
IAR DLIB Runtime Environment™	the DLIB runtime environment

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

Part I. Basic debugging

This part of the *C-SPY® Debugging Guide for RX* 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 EI/E20 and E2/E2 Lite/EZ-CUBE2 drivers
- The C-SPY J-Link driver

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.

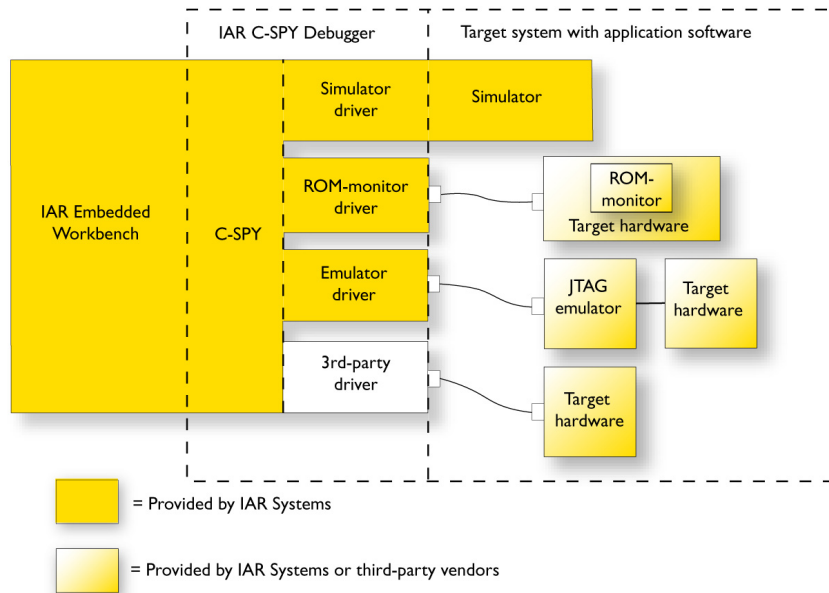
These topics are covered:

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

C-SPY AND TARGET SYSTEMS

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

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



Note: In IAR Embedded Workbench for RX, there are no ROM-monitor drivers.

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 39.

THE ROM-MONITOR PROGRAM

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

THIRD-PARTY DEBUGGERS

You can use a third-party debugger together with the IAR 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 the RX microcontrollers is available with drivers for these target systems and evaluation boards:

- Simulator
- E1 or E20 emulator
- E2 emulator
- E2 Lite/EZ-CUBE2 emulator
- J-Link debug probe.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

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

Feature	Simulator	E1	E20	E2	E2 Lite/ EZ-CUBE 2	J-Link
Code breakpoints*	Unlimited	Yes	Yes	Yes	Yes	Yes
Data breakpoints	Yes	Yes	Yes	Yes	Yes	Yes
Execution in real time	—	Yes	Yes	Yes	Yes	Yes
Zero memory footprint	Yes	Yes	Yes	Yes	Yes	Yes
Simulated interrupts	Yes	—	—	—	—	—
Real interrupts	—	Yes	Yes	Yes	Yes	Yes
Interrupt logging	Yes	—	—	—	—	—
Data logging	Yes	—	—	—	—	—
Live watch	Yes	Yes	Yes	Yes	Yes	Yes
Cycle counter	Yes	—	—	—	—	—
Code coverage	Yes	Yes	Yes	Yes	Yes	Yes
Data coverage*	Yes	—	Yes	—	—	—
Performance analysis*	—	Yes	Yes	Yes	Yes	Yes

Table 3: Driver differences

Feature	Simulator	E1	E20	E2	E2 Lite/ EZ-CUBE 2	J-Link
Start/stop routines	—	Yes	Yes	Yes	Yes	—
Profiling	Yes	Yes	Yes	Yes	Yes	Yes
Trace	Yes	Yes	Yes	Yes	Yes	Yes
Power debugging*	—	—	—	Limited	—	Yes

Table 3: Driver differences (Continued)

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

The IAR C-SPY Simulator

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

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.

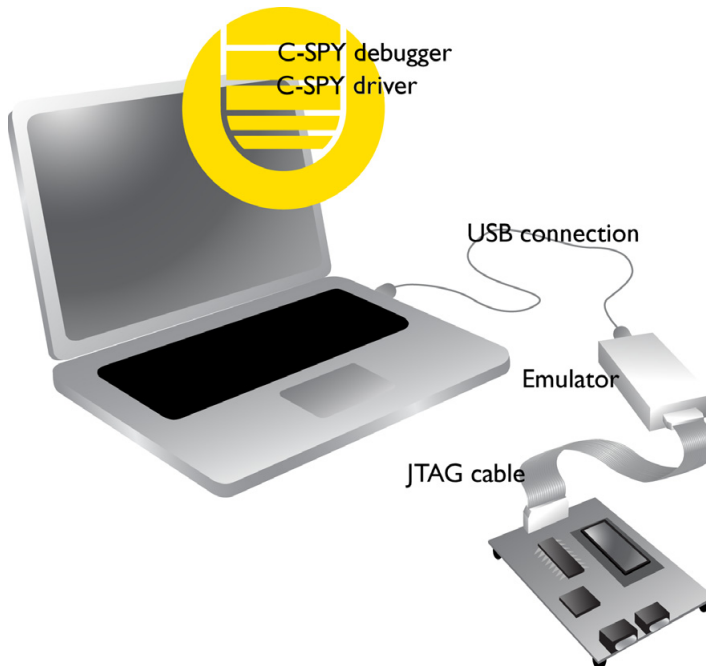
The C-SPY E1/E20 and E2/E2 Lite/EZ-CUBE2 drivers

C-SPY can connect to an E1, E2, E20, or E2 Lite/EZ-CUBE2 emulator using a C-SPY hardware debugger driver as an interface. The C-SPY hardware debugger drivers are automatically installed during the installation of IAR Embedded Workbench.

All RX microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

COMMUNICATION OVERVIEW

The C-SPY E1/E20 and E2/E2 Lite/EZ-CUBE2 drivers use USB to communicate with the emulator. The emulator communicates with the Front-end firmware (FFW) interface module. The FFW interface module, in turn, communicates with the Back-end firmware (BFW) module on the emulator.



For more information, see the documentation supplied with the emulator.

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

HARDWARE INSTALLATION

USB drivers are automatically installed during the installation of IAR Embedded Workbench. If you need to re-install them, they are available both on the installation CD and in the `rx\drivers\Renesas\` directory in the installation directory.

For more information about the hardware installation, see the documentation supplied with the E1, E2, E20, or E2 Lite or EZ-CUBE2 emulator from Renesas. The following power-up sequence is recommended to ensure proper communication between the target board, the emulator, and C-SPY:

- 1 Power up the target board.
- 2 Start the C-SPY debugging session.

The C-SPY J-Link driver

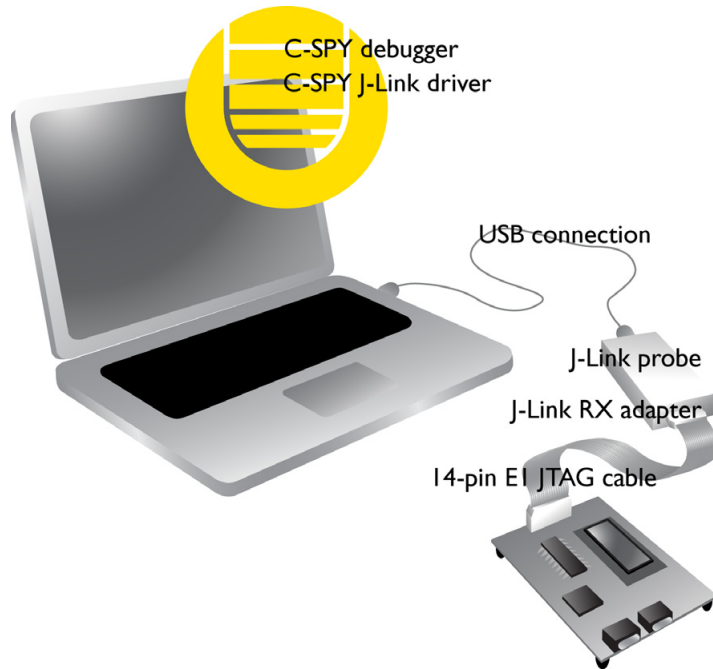
Using the C-SPY J-Link driver, C-SPY can connect to the J-Link debug probe. All RX microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

COMMUNICATION OVERVIEW

The C-SPY J-Link driver uses USB to communicate. There are two possible hardware configurations, depending on the target board:

- If the target board has a built-in J-Link, a USB cable connects the host computer directly to the target board.

- If you have a separate J-Link debug probe, the probe communicates with the JTAG interface on the microcontroller as in this figure:



For more information, see the documentation supplied with the J-Link debug probe or the target board.

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

HARDWARE INSTALLATION

USB drivers are automatically installed during the installation of IAR Embedded Workbench. If you need to re-install them, they are available both on the installation CD and in the `rx\drivers\JLink\` directory in the installation directory. For more information about the hardware installation, see the documentation supplied with the J-Link debug probe.

The following power-up sequence is recommended to ensure proper communication between the target board, debug probe, and C-SPY:

- 1 Power up the target board.

- 2** Power up the J-Link debug probe.
- 3** Start the C-SPY debugging session.

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 439.
- 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 RX*. See also *Adapting for target hardware*, page 53.

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 355.

For an example of how to use a setup macro file, see *Initializing target hardware before C-SPY starts*, page 54.

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 `rx\config` directory and they have the filename extension `.ddf`.

For more information about device description files, see *Adapting for target hardware*, page 53.

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 442.

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
- Downloading files to external flash memory
- Start debugging a running application.

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:

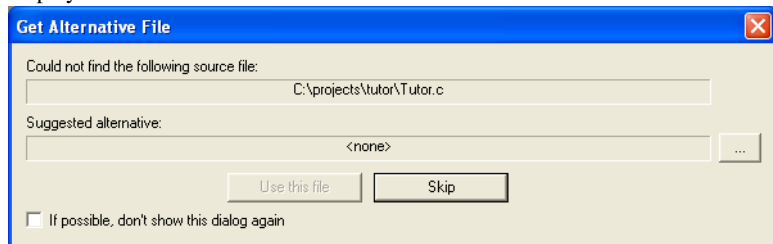
- 1 Choose **Project>Create New Project**, and specify a project name.
- 2 To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the file type drop-down list. Locate the executable file.
- 3 To start the executable file, click the **Download 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 60.

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 load additional debug images at C-SPY startup:

- 1 Choose **Project>Options>Debugger>Images** and specify up to three additional debug images to be loaded. For more information, see *Images*, page 441.
- 2 Start the debug session.

To load additional debug images at a specific moment:

Use the `__loadImage` system macro and execute it using either one of the methods described in *Using C-SPY macros*, page 357.

To display a list of loaded debug images:

Choose **Images** from the **View** menu. The **Images** window is displayed, see *Images window*, page 59.

EDITING IN C-SPY WINDOWS

You can edit the contents of these windows:

- **Memory** window
- **Symbolic Memory** window
- **Registers** window
- **Register User Groups Setup** window
- **Auto** window
- **Watch** window
- **Locals** window
- **Statics** window
- **Live Watch** window
- **Quick Watch** window

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

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

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

```
myArray;3
```

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

```
myPtr;3
```

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

```
myArray;5,10
```

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

```
myPtr;5,10
```

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

DOWNLOADING FILES TO EXTERNAL FLASH MEMORY

Normally, your application is downloaded to internal memory. To download the main executable file of your project to external flash memory, or any additional images that you have specified using the procedure described in *Loading multiple debug images*, page 49, you must configure C-SPY.

Note: Before following these instructions you must have defined external flash definition files (USD files) for your external flash memory, using the External Flash Definition Editor tool from Renesas Electronics Corporation. For more information, see the documentation from Renesas.

To configure C-SPY to load files to external flash memory:

- 1 Choose **C-SPY driver>Hardware Setup**.
- 2 On the **MCU** page of the dialog box, set the **Register setting** option to either **On-chip ROM enabled extended mode** or **On-chip ROM disabled extended mode**.
- 3 If you are using external RAM as working RAM, make sure that the **Byte order** option is set to the same byte order as the CPU.
- 4 On the **External Memory** page of the dialog box, specify up to four external flash definition files (USD files) by typing the absolute paths in the fields or by using the browse buttons. Note that only flash memory with 4096 or fewer sectors can be registered. If flash memory with more sectors is registered, programming cannot be guaranteed.
- 5 Decide whether to select the **Erase external flash ROM before download** options for the specified USD files. If the flash memory is not erased, addresses that are not overwritten by the download will keep their previous contents.

If your flash memory device does not support the Lock command (see the documentation from Renesas), select the **Erase external flash ROM before download** options for the specified USD files.

If you are allocating a device to multiple CS areas, do *not* select the **Erase external flash ROM before download** options for the specified USD files.

- 6** Now you are ready to download and debug, see *Starting a debug session*, page 48.

Note: Before downloading images to external flash memory, you must be aware of this:

- If the address ranges of multiple specified USD files overlap each other, connection with the hardware debugger cannot be established.
- For downloading to external flash memory, only internal RAM or CS area RAM can be used as a work area. SDRAM areas (SDCS) cannot be used as a work area.
- The work RAM area can also be used by your application, because the hardware debugger saves and restores data in this area. Note that the work RAM area cannot be specified either as the destination or origin of a DMA or DTC transfer, as an address where a DTC vector table or transfer information is to be allocated, or as the interrupt vector for a DMAC or DTC activation source.
- If your flash memory device does not support the Lock command (see the documentation from Renesas), USD files created with the External Flash Definition Editor tool from Renesas should be generated with the option **Clear Lock Bit** selected on the **USD File Creation** page.

See also *Hardware Setup dialog box: External Memory*, page 66.

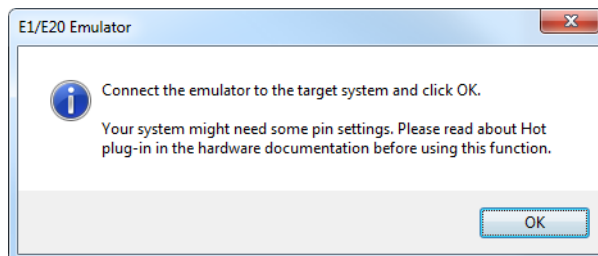
START DEBUGGING A RUNNING APPLICATION

Using an E1, E20, E2, or E2 Lite/EZ-CUBE2 emulator, you can start debugging a running application at its current location, without resetting the target system.

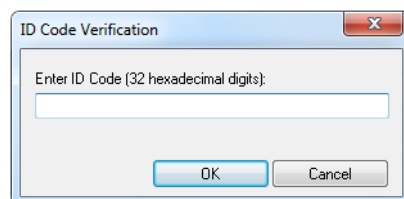
Start debugging from the middle of execution

- 1** Make sure that your application is running on the target board and that the target board is powered by external power.
- 2** Choose **Project>Attach to Running Target**. For information about this menu command, see the *IDE Project Management and Building Guide for RX*.

- 3 When you are prompted, connect the emulator to the target board and click OK.



- 4 Enter the ID code of the target MCU in the **ID Code Verification** dialog box.



- 5 When the debug session starts, your application is still executing but now you can monitor RAM and look at variables in the **Live Watch** window.
- 6 To stop execution, click **Stop** or set a breakpoint.
- 7 You can now debug your application as usual.

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 47. Device description files contain device-specific information such as:

- Memory information for device-specific memory zones, see *C-SPY memory zones*, page 153.

- Definitions of memory-mapped peripheral units, device-specific CPU registers, and groups of these.
- Definitions for device-specific interrupts, which makes it possible to simulate these interrupts in the C-SPY simulator, see *Interrupts*, page 331.
- The device name and the MCU filename, used by emulators

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

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

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

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 55
- *Images window*, page 59
- *Get Alternative File dialog box*, page 60
- *Download Emulator Firmware dialog box*, page 61
- *Operating Frequency dialog box*, page 62
- *Hardware Setup dialog box: MCU*, page 63
- *Hardware Setup dialog box: External Memory*, page 66
- *External Area dialog box*, page 68

See also:

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

C-SPY Debugger main window

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

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

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

Menu bar

These menus are available during a debug session:

Debug

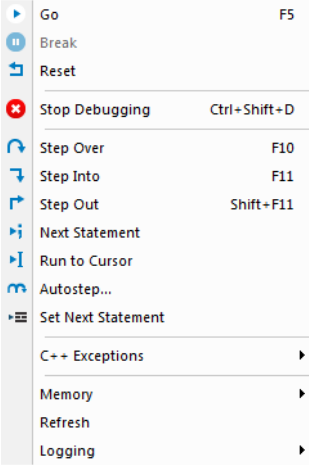
Provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

C-SPY driver menu

Provides commands specific to a C-SPY driver. The driver-specific menu is only available when the driver is used. For information about the driver-specific menu commands, see *Reference information on C-SPY driver menus*, page 449.

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.



Break

Stops the application execution.



Reset

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

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

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



Stop Debugging (Ctrl+Shift+D)

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



Step Over (F10)

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



Step Into (F11)

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



Step Out (Shift+F11)

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



Next Statement

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



Run to Cursor

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



Autostep

Displays a dialog box where you can customize and perform autostepping, see *Autostep settings dialog box*, page 93.



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 163.

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 164.

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 87.

C-SPY windows

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

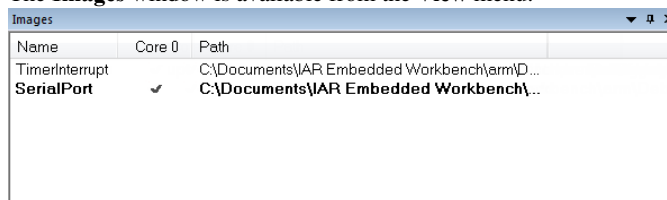
- C-SPY Debugger main window
- Disassembly window

- 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 217
- 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\JAR Embedded Workbench\am\D...
SerialPort	✓	C:\Documents\JAR 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 49.

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.

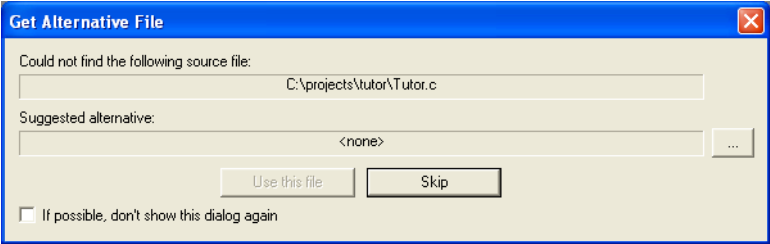
Related information

For related information, see:

- *Loading multiple debug images*, page 49
- *Images*, page 441
- *__loadImage*, page 380

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 48.

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

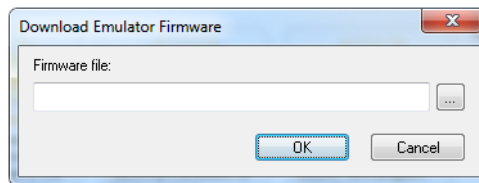
Instead of displaying the dialog box again for a missing source file, C-SPY will use the previously supplied response.

Related information

For related information, see *Starting a debug session with source files missing*, page 48.

Download Emulator Firmware dialog box

The Download Emulator Firmware dialog box is available from the **C-SPY driver** menu.



Use this dialog box to update the firmware of your emulator if needed. The only reason for doing this manually is if the automatically downloaded firmware is not working correctly.

Requirements

One of these alternatives:

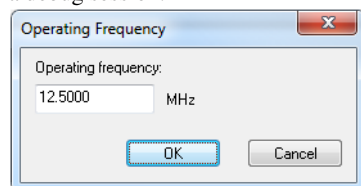
- The C-SPY E1/E20 driver
- The C-SPY E2/E2 Lite/EZ-CUBE2 driver.

Firmware file

Browse to the firmware file on your host computer and click **OK** to download it to your emulator hardware. The emulator firmware files have the filename extension `.s` and are located in subdirectories of the `rx\external\` directory of your product installation.

Operating Frequency dialog box

The **Operating Frequency** dialog box is available from the **C-SPY driver** menu during a debug session.



Use this dialog box to inform the emulator of the operating frequency that the MCU is running at. This information is used by the **Timeline** window and by performance counters to convert clock cycles into time.

Requirements

A C-SPY hardware debugger driver.

Operating frequency

Specifies the operating frequency that the MCU is running at. This value is used by the performance analysis to convert cycles to time and by the J-Link **Timeline** window to estimate the number of elapsed cycles.

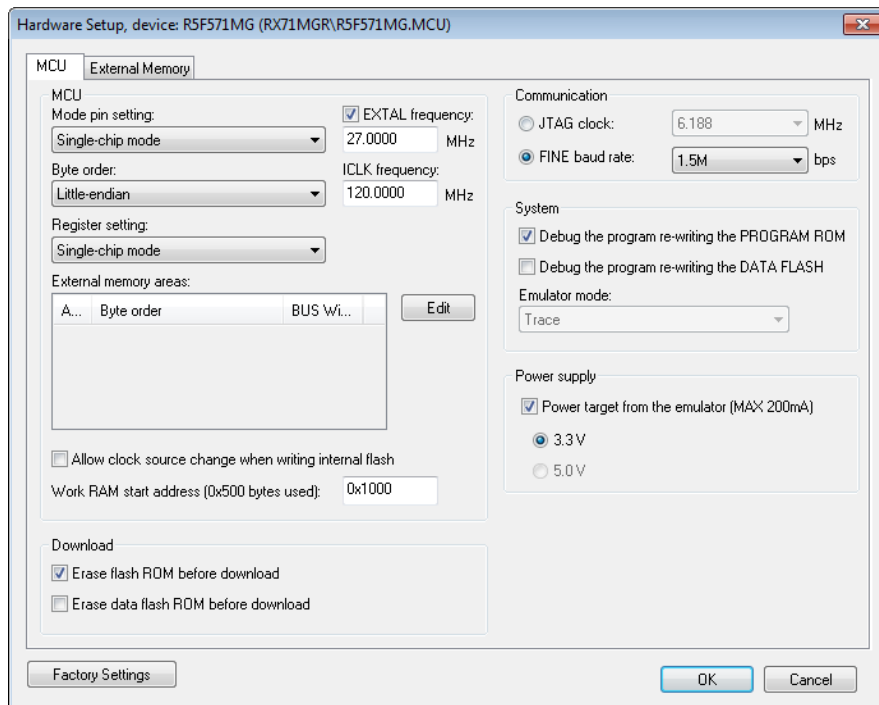
Related information

For related information, see:

- *Performance Analysis Setup dialog box*, page 257
- *The application timeline*, page 211.

Hardware Setup dialog box: MCU

The **Hardware Setup** dialog box for the hardware debuggers is available from the **C-SPY driver** menu. Before C-SPY is started for the first time in a new project, and when you change devices, the hardware must be configured.



Use the **MCU** options page to make general settings that control how the emulator operates.

Requirements

A C-SPY hardware debugger driver.

Mode pin setting

Controls the MCU operation based on the pin settings. Choose between:

- Single-chip mode
- User boot mode
- USB boot mode.

Note: Not all modes are available for all devices or all hardware debuggers.

EXTAL frequency

Specify the frequency in MHz of the external clock source that supplies the target MCU.

ICLK frequency

Specify the frequency in MHz of the internal clock source.

Note: This option is not available for all devices or all hardware debuggers.

Byte order

Controls the byte order of the device. Choose between:

- Little-endian
- Big-endian.

Note: This option is not available for all devices or all hardware debuggers.

Register setting

Controls the MCU operation based on register settings. Choose between:

- Single-chip mode
- On-chip ROM enabled extended mode
- On-chip ROM disabled extended mode.

Note: Not all modes are available for all devices or all hardware debuggers.

External memory areas

Lists the defined external memory areas. To edit a memory area, select the area and click **Edit** to display the **External Area** dialog box, see *External Area dialog box*, page 68.

Area

The name of the external memory area.

Byte order

Identifies whether the byte order is the same as the byte order of the MCU or different.

BUS width

The bus width of the area: 8, 16 or 32 bits.

No external memory areas are defined if the **Register setting** is **Single-chip mode**.

Allow clock source change when writing internal flash

Allows the clock source to change while internal flash memory is being rewritten in the emulator.

Work RAM start address

Specify the start address of the working RAM area for the debugger. The specified amount of bytes, beginning with the start address you specify, is used by the emulator firmware. The debugger uses the memory area when programming the on-chip flash memory, so the working RAM must be within the on-chip RAM area.

Your application can also use this area (because memory data in this area will be saved on the host computer and then restored), but do not specify any address in this area as the origin or destination of a transfer by the DMA or DTC.

Erase flash ROM before download

Erases the (internal) flash ROM before your application is downloaded. If this option is deselected, the flash ROM memory will not be erased by the downloading process. This means that any addresses that are not overwritten by the downloaded image will keep their previous contents.

Note: If multiple images are downloaded, you must deselect this option.

Erase data flash ROM before download

Erases the (internal) data flash ROM before your application is downloaded. If this option is deselected, the data flash ROM memory will not be erased by the downloading process. This means that any addresses that are not overwritten by the downloaded image will keep their previous contents.

Communication

Controls the communication between the emulator and the host computer. Choose between:

JTAG clock

Selects the JTAG interface. Choose a communication clock frequency.

FINE baud rate

Selects the FINE single wire debug interface. Choose a communication speed in bits/second.

Note that this option is not available for all devices or all hardware debuggers.

Debug the program re-writing the PROGRAM ROM

Debugs the program which writes to the program ROM (flash memory).

Debug the program re-writing the DATA FLASH

Debugs the program which writes to the data flash memory.

Emulator mode

Controls how the hardware debugger can be used.

Trace

Makes the trace functionality of the C-SPY driver available, see *Collecting and using trace data*, page 190.

Power target from the emulator

Select this option and the correct voltage if you are supplying the target board with power from the hardware debugger, and not from an external power supply.

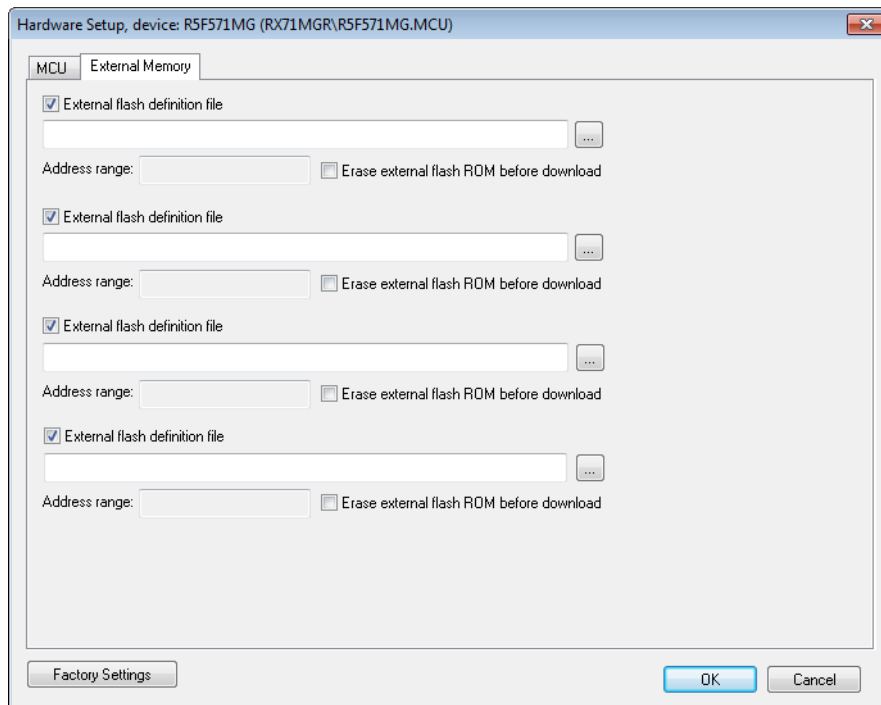
If you select this option but connect an external power supply to the target board, the external power supply will be used instead and these settings will be ignored.

Note: This option is not available for all devices or all hardware debuggers.

Hardware Setup dialog box: External Memory

The **Hardware Setup** dialog box for the hardware debuggers is available from the **C-SPY driver** menu. Before C-SPY is started for the first time in a new project, and

when you change devices, you can specify how your application will be downloaded to the external flash memory.



Use the **External Memory** options page to specify how your application will be downloaded to the external flash memory.

Using these options, you can download the main output file of your project or the additional debug images specified on the **Project>Options>Debugger>Images** page to flash memory connected to an external bus.

Requirements

A C-SPY hardware debugger driver.

External flash definition file

Specify an external flash definition file (USD file) by typing the absolute path to the file or by using the browse button to navigate to the file. Up to four USD files can be registered.

For more information about USD files, see the External Flash Definition Editor documentation on www.renesas.com.

Address range

The address range of the download defined in the specified USD file.

Erase external flash ROM before download

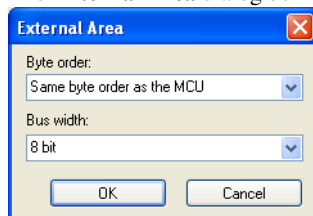
Erases the external flash ROM before your application is downloaded. If this option is left deselected, the flash ROM memory will not be erased by the downloading process. This means that any addresses that are not overwritten by the downloaded image will keep their previous contents.

Related information

For related information, see *Downloading files to external flash memory*, page 51.

External Area dialog box

The **External Area** dialog box is available from the **Hardware Setup** dialog box.



Use this dialog box to edit a defined external memory area, see *Hardware Setup dialog box: MCU*, page 63.

Requirements

A C-SPY hardware debugger driver.

Byte order

Controls the byte order of the memory area. Choose between:

- **Same byte order as the MCU**
- **Different byte order from the MCU.**

Bus width

The bus width of the area. Choose between:

- **8 bit**

- 16 bit
- 32 bit.

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 74 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 93.

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 127 and *Breakpoint consumers*, page 127.

- 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 154.
- 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.



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.

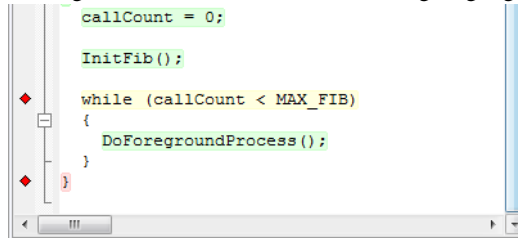
Code coverage

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 265.

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:



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 Reference Guide for RX*.

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 86 and *Terminal I/O Log File dialog box*, page 87.

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 88. 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

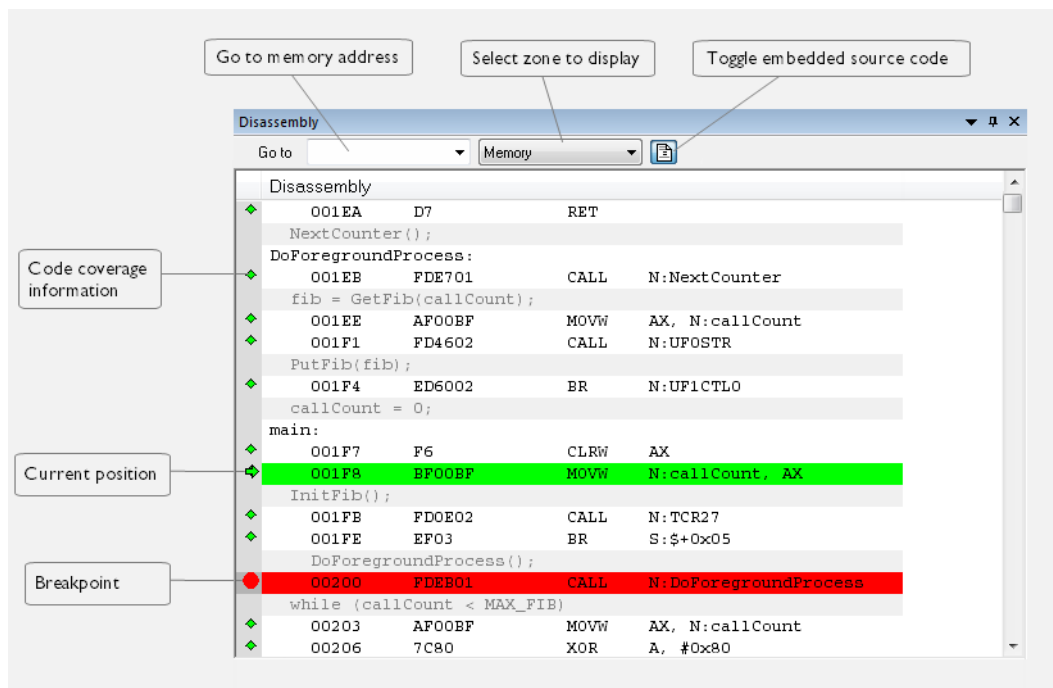
Reference information about:

- *Disassembly window*, page 79
- *Call Stack window*, page 84
- *Terminal I/O window*, page 86
- *Terminal I/O Log File dialog box*, page 87
- *Debug Log window*, page 88
- *Report Assert dialog box*, page 89
- *Start/Stop Function Settings dialog box*, page 90
- *Select Label dialog box*, page 92
- *Autostep settings dialog box*, page 93
- *ID Code Verification dialog box*, page 93.
- *Cores window*, page 94

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

Disassembly window

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



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

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

- 1 Choose **Tools>Options>Debugger**.
- 2 Set the default color using the **Source code 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 71.

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 153.

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

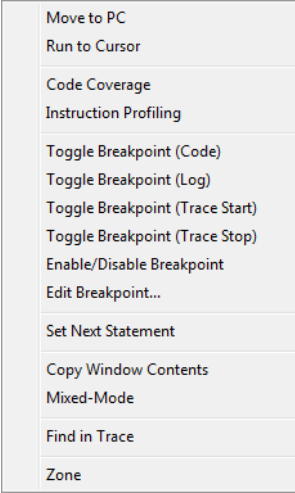
The display area shows the disassembled application code. This area contains these graphic elements:

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 123.
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.

Context menu

This context menu is available:



Note: The contents of this menu are dynamic, which means that the commands on the menu depend on the C-SPY driver.

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 137.

Toggle Breakpoint (Hardware Code)

Toggles a hardware code breakpoint. Assembler instructions and any corresponding label at which hardware code breakpoints have been set are highlighted in red. Note that this menu command is only available for C-SPY hardware debugger drivers. For more information, see *Hardware Code Breakpoint dialog box*, page 138.

Toggle Breakpoint (Software Code)

Toggles a software code breakpoint. Assembler instructions and any corresponding label at which software code breakpoints have been set are highlighted in red. Note that this menu command is only available for C-SPY hardware debugger drivers. See *Software Code Breakpoint dialog box*, page 140.

Toggle Breakpoint (Performance Start)

Toggles a Performance Start breakpoint. If the breakpoint has been selected in the **Performance Analysis Setup** dialog box, the performance analysis starts when this breakpoint is triggered. Note that this menu command is only available if the C-SPY driver you are using supports performance analysis. See *Performance Start breakpoints dialog box*, page 261.

Toggle Breakpoint (Performance Stop)

Toggles a Performance Stop breakpoint. If the breakpoint has been selected in the **Performance Analysis Setup** dialog box, the performance analysis stops when this breakpoint is triggered. Note that this menu command is only available if the C-SPY driver you are using supports performance analysis. See *Performance Stop breakpoints dialog box*, page 262.

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 141.

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 205.

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 206.

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

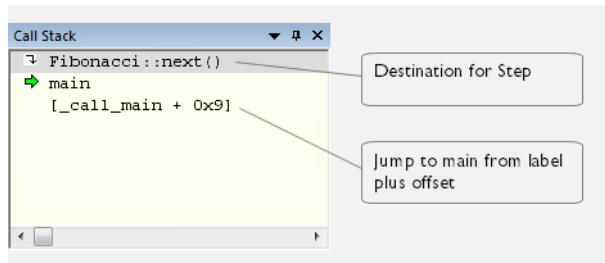
Searches the contents of the **Trace** window for occurrences of the given location—the position of the insertion point in the source code—and reports the result in the **Find in Trace** window. This menu command requires support for Trace in the C-SPY driver you are using, see *Differences between the C-SPY drivers*, page 39.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 153.

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 77.

Requirements

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

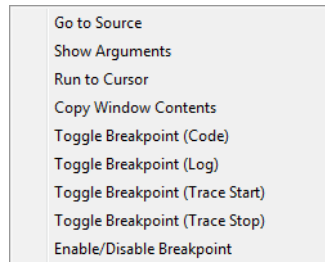
Display area

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

<code>function(values)***</code>	A C/C++ function with debug information. Provided that Show Arguments is enabled, <i>values</i> 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 <i>IAR C/C++ Development Guide for RX</i> .
<code>[label + offset]</code>	An assembler function, or a C/C++ function without debug information.
<code><exception_frame></code>	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)

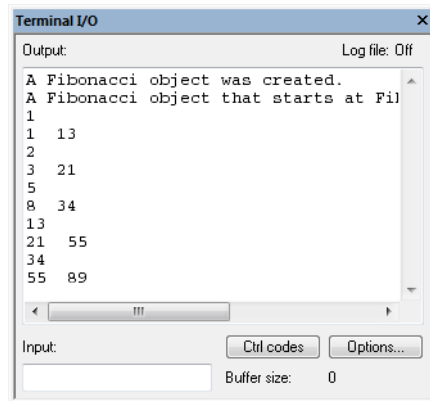
Toggles a Trace Stop breakpoint. When the breakpoint is triggered, trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports it.

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 77.

Requirements

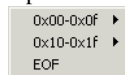
Can be used with all C-SPY debugger drivers and debug probes.

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.

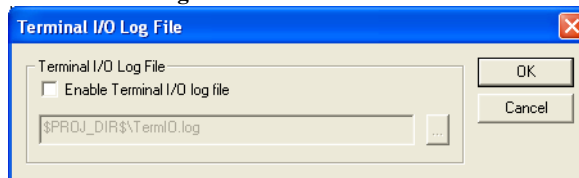


Options

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 RX*.

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 77.

Requirements

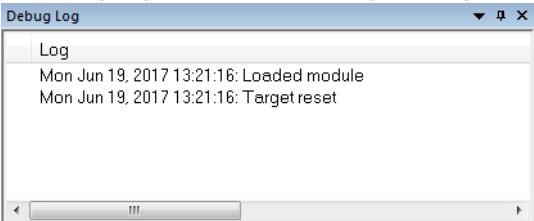
Can be used with all C-SPY debugger drivers and debug probes.

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 RX*.

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>
```

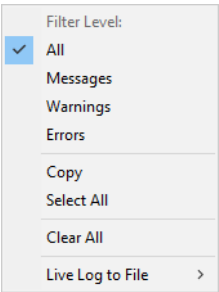
See also *Debug logging*, page 78.

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

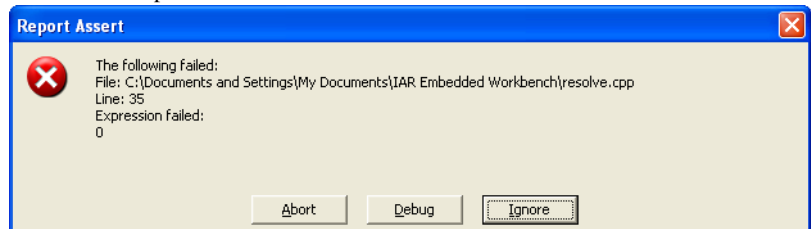
Clears the contents of the window.

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

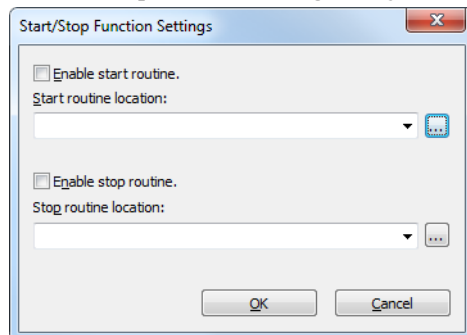
C-SPY stops the execution of the application and returns control to you.

Ignore

The assertion is ignored and the application continues to execute.

Start/Stop Function Settings dialog box

The **Start/Stop Function Settings** dialog box is available from the C-SPY *Driver* menu.



Use this dialog box to configure the emulator to execute specific routines of your application immediately before the execution starts and/or after it halts. This is useful if you want to control your system in synchronization with starting and stopping the execution of your application.

Requirements

One of these alternatives:

- The C-SPY E1/E20 driver
- The C-SPY E2/E2 Lite/EZ-CUBE2 driver.

Restrictions on using start/stop routines

Some restrictions apply:

- When the start/stop feature is enabled you cannot:
 - let your application use the RAM area 0x0–0x22F. For example, change the start address for RAM_region16, RAM_region24, and RAM_region32 from 0x4 to 0x230 in the linker configuration file.
 - set memory or download into the program area of a start/stop routine
 - set breakpoints in the program area of a start/stop routine.
- While either of the start/stop routines is running, the four bytes of memory indicated by the interrupt stack pointer are in use by the emulator.
- In the start/stop routines, these restrictions apply to registers and flags:

Register and flag names	Restrictions
ISP register	When execution of a start/stop routine is ended, the register must be returned to its value at the time the routine started.
Flag U	While a specified start/stop is running, switching to user mode is prohibited.
Flag I	No interrupts are allowed during execution of a start/stop routine.
Flag PM	While a start/stop routine is running, switching to user mode is prohibited.

Table 4: Restrictions on registers and flags

- When either of the start/stop routines is running, the following does not work:
 - Trace
 - Breaks in execution in the start/stop routines
 - Performance measurement. The start/stop routines are not within the scope of performance measurement.
 - Events. Event settings are invalid within the start/stop routines.
- While either of the start/stop routines is running, non-maskable interrupts are always disabled.

This table shows which state the MCU will be in when your application starts running after executing a start routine:

MCU resource	Status
MCU general purpose registers	These registers are in the same state as when your application last stopped, or in states determined by the settings in the Registers window. Changes made by the start routine to the contents of registers are not reflected.
Memory in the MCU space	Accesses to memory after the start routine has finished executing are reflected.
MCU peripheral functions	The operation of the MCU's peripheral functions is continued after the start routine has finished executing.

Table 5: MCU status when the user application starts executing

Enable start routine

Enables the execution of a routine immediately before your application starts executing.

Start routine location

Specifies the routine to be executed immediately before your application starts executing. Type a label or an address, or click the browse button to open the **Select Label** dialog box; see *Select Label dialog box*, page 92.

Enable stop routine

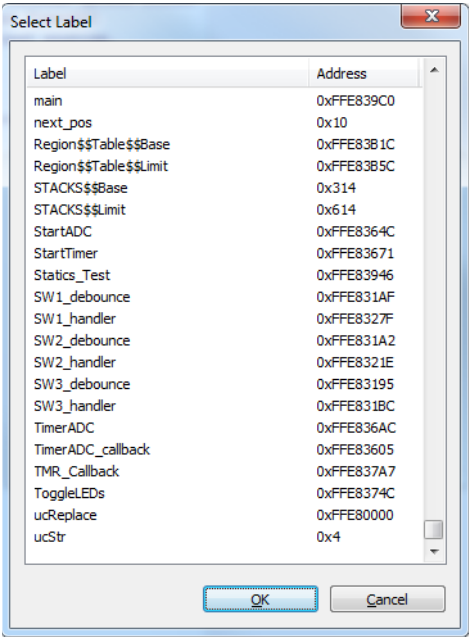
Enables the execution of a routine immediately after your application stops executing.

Stop routine location

Specifies the routine to be executed immediately after your application stops executing. Type a label or an address, or click the browse button to open the **Select Label** dialog box; see *Select Label dialog box*, page 92.

Select Label dialog box

The **Select Label** dialog box is available from the **Start/Stop Function Settings** dialog box.



Select the routine you want to be executed and click **OK**.

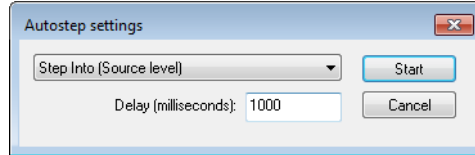
Requirements

One of these alternatives:

- The C-SPY E1/E20 driver
- The C-SPY E2/E2 Lite/EZ-CUBE2 driver.

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 72. You can stop the autostepping by clicking the Break button on the debug toolbar.

Requirements

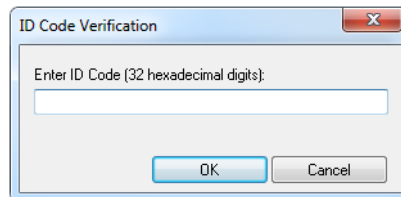
Can be used with all C-SPY debugger drivers and debug probes.

Delay (milliseconds)

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

ID Code Verification dialog box

The **ID Code Verification** dialog box is displayed if the ID code programmed in the target MCU differs from the ID code in the application that is being downloaded.




If this dialog box is displayed, verify that you have the right to debug or download your application to the target board.

Enter ID Code

Specify the ID code of the target MCU, a sequence of 32 hexadecimal digits.

Cores window

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

Cores				
Core	Status	PC	Cycles	
 0: Core 0	Stopped	0x00055F	74	

This window displays information about the executing core, such as its execution state. This information is primarily useful for IAR Embedded Workbench products that support multicore debugging.

Requirements

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

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.

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.
- The **Data Sample** window displays samples for up to four different variables. You can also display the data samples as graphs in the **Sampled Graphs** window. By using data sampling, you will get an indication of the data value over a length of time. Because it is a sampled value, data sampling is best suited for slow-changing data.

For more information about these windows, see *The application timeline*, page 211.

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 53.

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

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

Table 6: C-SPY assembler symbols expressions

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

Example	What it does
#PC	Refers to the program counter.
#`PC`	Refers to the assembler label PC.

Table 7: Handling name conflicts between hardware registers and assembler labels

Which processor-specific symbols are available by default can be seen in the **Registers** window, using the CPU Registers register group. See *Registers window*, page 173.

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 356.

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 362.

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 213.

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.

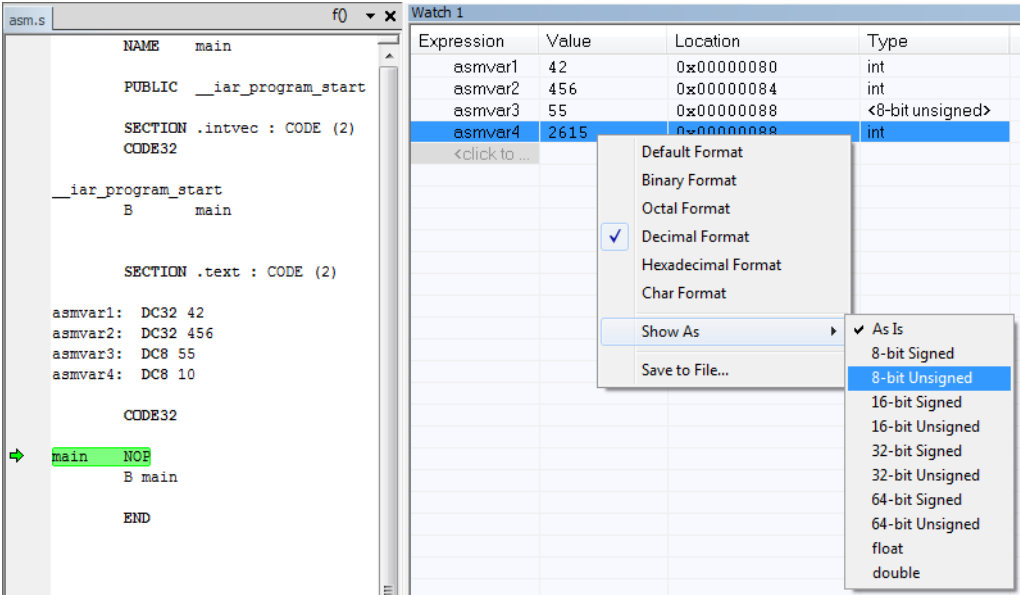
Right-click in any of the windows to access the context menu which contains additional commands. Convenient drag-and-drop between windows is supported, except for in the

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:



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 103
- *Locals window*, page 106
- *Watch window*, page 108
- *Live Watch window*, page 111
- *Statics window*, page 113
- *Quick Watch window*, page 116
- *Symbols window*, page 119
- *Resolve Symbol Ambiguity dialog box*, page 122

See also:

- *Reference information on trace*, page 192 for trace-related reference information
- *Macro Quicklaunch window*, page 412

Auto window

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

Auto				
Expression	Value	Location	Type	
NextCounter	NextCounter {0x40B}		void (___...	
fib	1	Memory: 0xFE74	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 50.

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

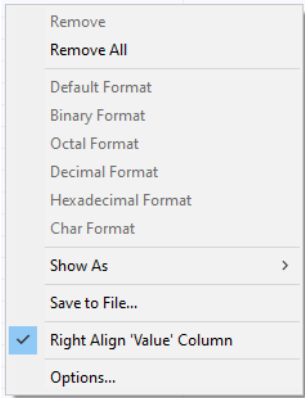
The location in memory where this variable is stored.

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 102.

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.

Locals			
Variable	Value	Location	Type
i	1244	Memory : 0xFEF72	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 50.

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

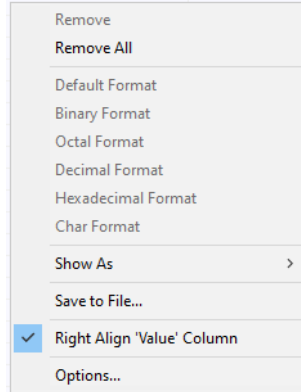
The location in memory where this variable is stored.

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 102.

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.

Watch 1			
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.

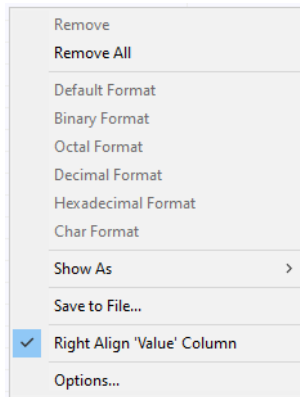
See also *Editing in C-SPY windows*, page 50.

Requirements

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

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 102.

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.

Live Watch				
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 50.

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

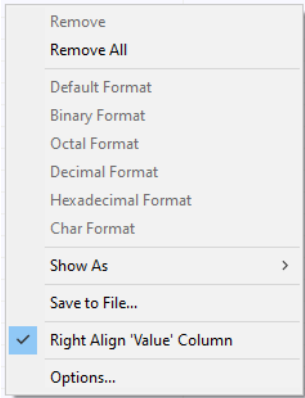
The location in memory where this variable is stored.

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 102.

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	
_Myfirst	0xA0E0	Memory : 0xFB134	class _Vector_value<allocator...	
_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 50.

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:

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

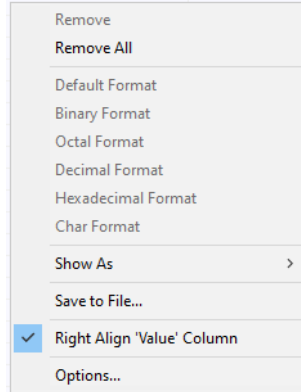
The location in memory where this variable is stored.

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 102.

Save to File

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

Right Align ‘Value’ Column

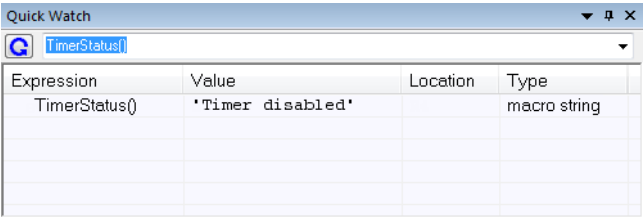
Right-aligns the contents of the **Value** column.

Options

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

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 50.

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 357.

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

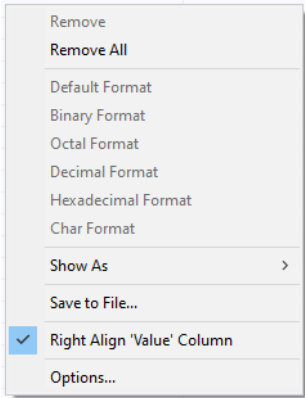
The location in memory where this variable is stored.

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 102.

Save to File

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

Right Align ‘Value’ Column

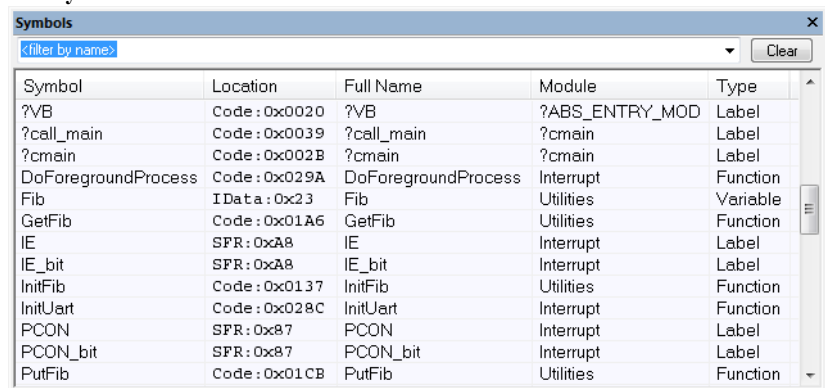
Right-aligns the contents of the **Value** column.

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.

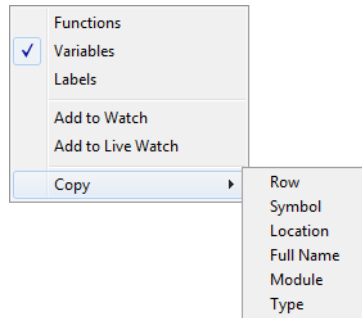
Type

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.

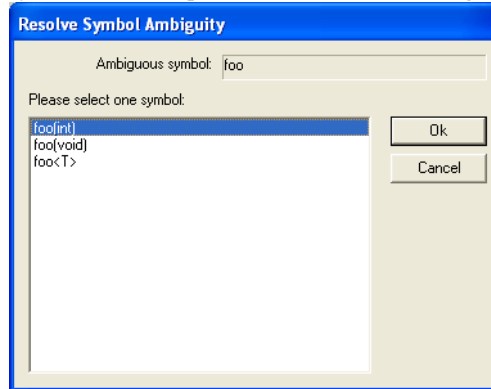
Full Name Copies the contents of the **Full Name** cell on the selected line.

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 127.

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 72.

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.

Note: When you use a C-SPY hardware debugger driver and set a breakpoint in code without specifying the type, a *hardware code* breakpoint will be set as long as there are any available. If there are no available hardware code breakpoints, a *software code* breakpoint will be set.

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 trace collection breakpoints

Data trace collection breakpoints are useful for collecting trace information from variables that have a fixed address in memory. Trace information is collected every time that data is accessed at the specified location. This does not stop the execution.

Data breakpoints

Data breakpoints are primarily useful for variables that have a fixed address in memory. If you set a breakpoint on an accessible local variable, the breakpoint is set on the corresponding memory location. The validity of this location is only guaranteed for small parts of the code. Data breakpoints are triggered when data is accessed at the specified location.

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

Data Log breakpoints

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

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

Using a single instruction, the microcontroller can only access values that are 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.

Hardware code breakpoints

Hardware code breakpoints are code breakpoints that use hardware resources. Because memory does not have to be reprogrammed after a hardware code breakpoint has been hit, hardware code breakpoints are preferred over software code breakpoints. However, the number of available hardware breakpoints is limited, see *Breakpoints in the C-SPY hardware debugger drivers*, page 127.

Software code breakpoints

Software code breakpoints are code breakpoints that use software resources. After a software code breakpoint has been triggered the memory must be reprogrammed, so software code breakpoints should if possible be used only in RAM memory or for breakpoints that are not triggered so often.

Performance breakpoints

By default, performance analysis is running from when the hardware debugger starts until it stops. Performance Start and Stop breakpoints are used for analyzing the performance over a smaller region of code. For reference information about these breakpoints, see *Reference information on performance analysis*, page 256.

Performance Start and Stop breakpoints share the same resources as hardware code breakpoints, see *Breakpoints in the C-SPY hardware debugger drivers*, page 127.

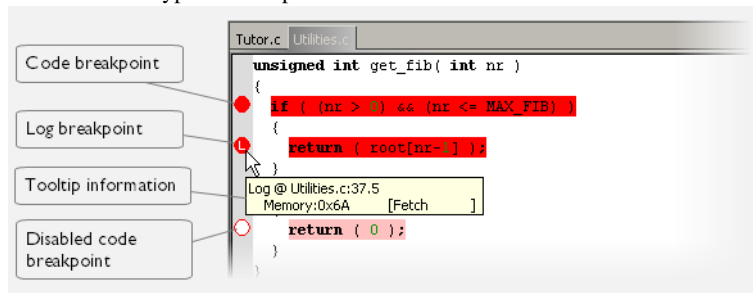
Immediate breakpoints

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 RX*.



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. If no hardware breakpoints are available, *software breakpoints* will be used.

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

C-SPY hardware driver	Software code breakpoints	Hardware code and Log breakpoints	Trace breakpoints	Data breakpoints
E1/E2/E20/E2 Lite/EZ-CUBE2				
using 12 hardware breakpoints*	256	Up to 8†	Up to 8†	Up to 4*
J-Link				
using 12 hardware breakpoints*	256	Up to 8†	Up to 8†	Up to 4*

Table 8: Available breakpoints in C-SPY hardware debugger drivers

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

† These 8 breakpoint resources are shared between three types of breakpoints.

Hardware breakpoints in the RX emulators share the same resources. There is a total of 12 of these breakpoint events, divided into 8 that are triggered by a program counter access and 4 that are triggered by a data access.

The *software code* breakpoints use a mechanism that writes to the memory with a `BRK` instruction, and when a breakpoint is triggered the original instruction is written back to the memory. This makes it possible to use the breakpoint for code in RAM, but if used for code in flash memory the execution is slowed down by the need to reprogram the memory. There are up to 256 software code breakpoints.

The debugger will first use any available hardware breakpoints before using software breakpoints.

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:

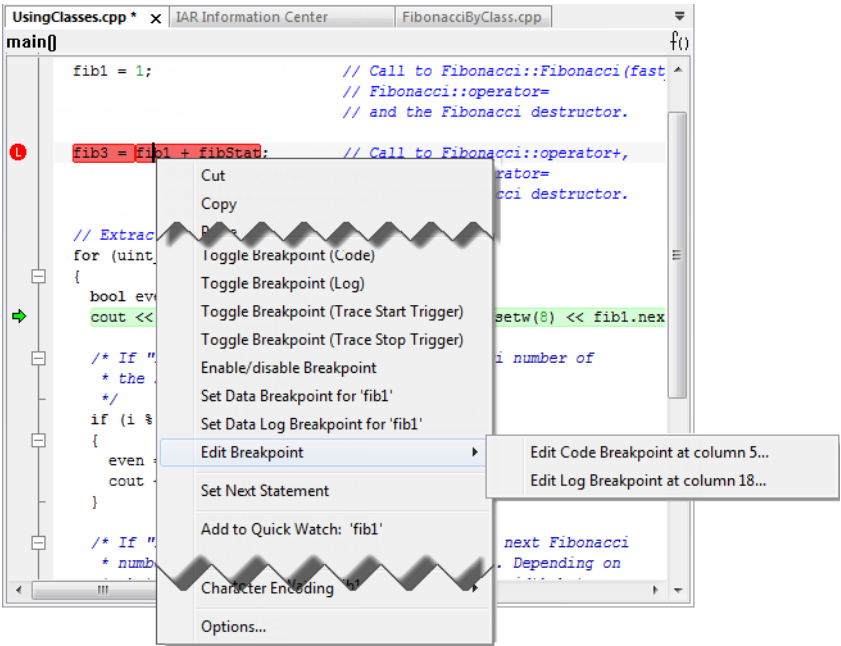
- I Choose **View>Breakpoints** to open the **Breakpoints** window.

- 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	E1/E20	E2/E2 Lite/ EZ-CUBE2	J-Link
__setCodeBreak	Yes	Yes	Yes	Yes
__setDataBreak	Yes	—	—	—
__setLogBreak	Yes	Yes	Yes	Yes
__setSimBreak	Yes	—	—	—
__setTraceStartBreak	Yes	Yes	Yes	Yes
__setTraceStopBreak	Yes	Yes	Yes	Yes
__clearBreak	Yes	Yes	Yes	Yes

Table 9: C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 369.

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 357.

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 134
- *Breakpoint Usage window*, page 136
- *Code breakpoints dialog box*, page 137
- *Hardware Code Breakpoint dialog box*, page 138
- *Software Code Breakpoint dialog box*, page 140
- *Log breakpoints dialog box*, page 141
- *Data breakpoints dialog box (Simulator)*, page 142
- *Data breakpoints dialog box (C-SPY hardware debugger drivers)*, page 144

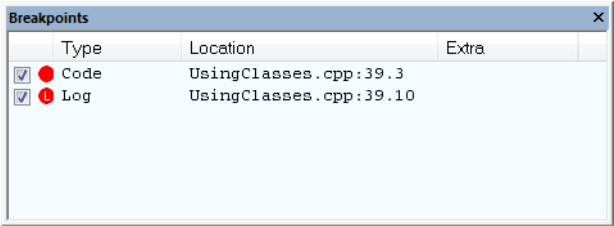
- *Data Log breakpoints dialog box*, page 146
- *Immediate breakpoints dialog box*, page 147
- *Enter Location dialog box*, page 148
- *Resolve Source Ambiguity dialog box*, page 150

See also:

- *Reference information on C-SPY system macros*, page 369
- *Reference information on trace*, page 192
- *Data Trace Collection breakpoints dialog box*, page 207
- *Reference information on performance analysis*, page 256

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

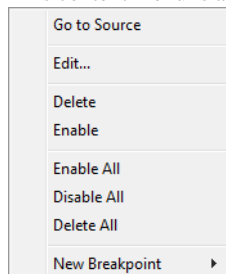
Can be used with all C-SPY debugger drivers and debug probes.

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

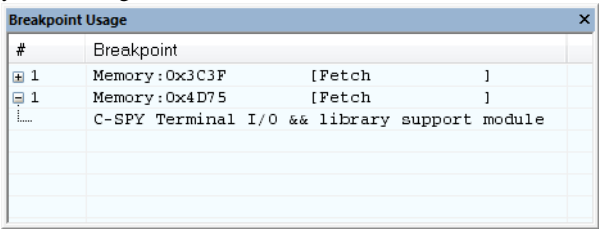
Deletes all defined breakpoints.

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. 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 127.

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.

The screenshot shows the 'Code' breakpoints dialog box. It has a title bar with a red dot and the word 'Code'. The main area is divided into several sections. The 'Break at:' section has a text box and an 'Edit...' button. The 'Size' section has two radio buttons: 'Auto' (selected) and 'Manual', and a text box with the value '0'. The 'Action' section has an 'Expression:' text box. The 'Conditions' section has an 'Expression:' text box. At the bottom, there are two radio buttons: 'Condition true' (selected) and 'Condition changed', and a 'Skip count:' text box with the value '0'.

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 129.

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 148.

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 is only available for the C-SPY simulator driver.

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 132.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 98.

Condition true

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

Condition changed

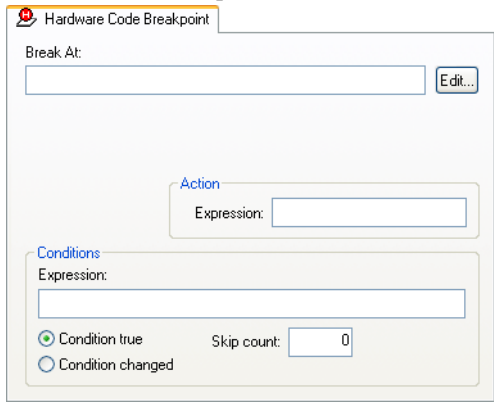
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

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.

Hardware Code Breakpoint dialog box

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



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

Requirements

A C-SPY hardware debugger driver.

Break At

Specify the location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 148.

Action

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

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 98.

Condition true

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

Condition changed

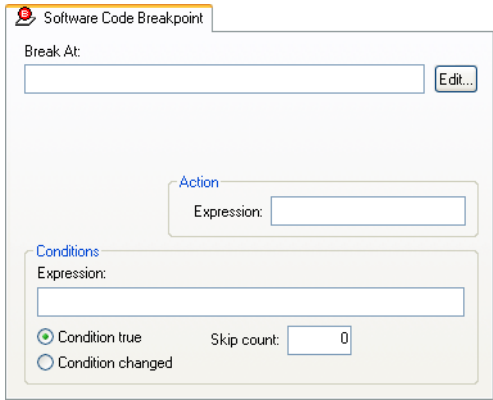
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

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.

Software Code Breakpoint dialog box

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



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

Note: Because the memory must be reprogrammed after a software code breakpoint has been triggered, software code breakpoints should if possible be used only in RAM memory or for breakpoints that are not triggered so often.

Requirements

A C-SPY hardware debugger driver.

Break At

Specify the location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 148.

Action

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

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 98.

Condition true

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

Condition changed

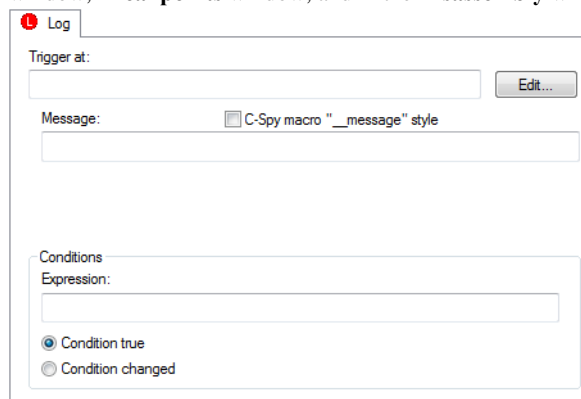
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

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 129.

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 148.

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 365.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 98.

Condition true

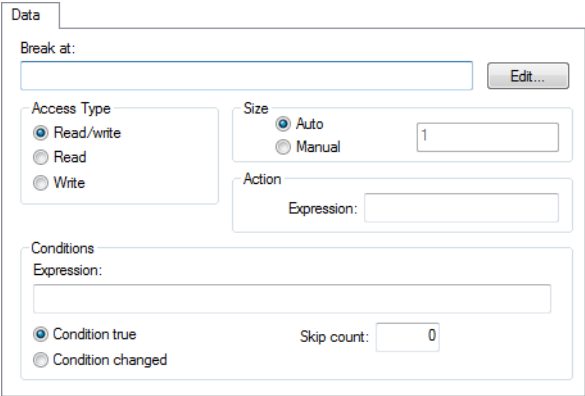
The breakpoint is triggered if the value of the expression is true.

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 129. 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 148.

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 132.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 98.

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 (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.

This figure reflects the C-SPY hardware debugger drivers.

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

Requirements

A C-SPY hardware debugger driver.

Break At

Specify the location of the breakpoint, or the start location if you select the address condition **Range**. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 148.

Address Condition

Determines whether there should be a trigger condition for the address where the data is located:

None

There is no condition for the address.

Mask

Specify an address mask and choose a **Compare** setting.

Compare

Specify whether the condition is that the address should be equal (Specified value (==)) or not equal (Any other value (!=)) to the mask.

Range

Uses the location in the **Break At** field as the start address of an address range. Use the text box to specify whether the trigger condition applies to inside the range or outside the range.

End address

Specify the end location if the **Break At** field contains the start location of an address range. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 148

Data Condition

Determines whether there should be data trigger conditions for the breakpoint:

Read/Write

Read — the breakpoint is only triggered by a read access

Write — the breakpoint is only triggered by a write access

Read/Write — the breakpoint is triggered by all accesses.

Access 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. Choose between:

Auto — The access size will be set automatically.

Byte — The access size will be set to a byte.

Word — The access size will be set to a word.

Long — The access size will be set to a long.

Compared data

Set a value that the accessed data should be compared to, using decimal notation or hexadecimal notation (prefixed by 0x).

Mask

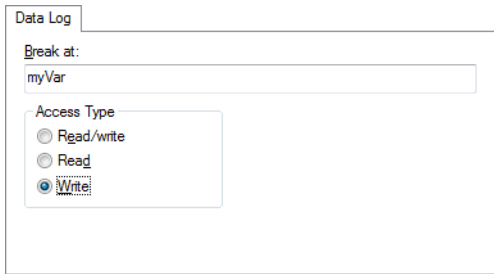
Set a mask for the compared data.

Compare

Specify whether the condition is that the data should be equal (Specified value (==)) or not equal (Any other value (!=)) to the mask.

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 129.

See also *Data Log breakpoints*, page 125 and *Getting started using data logging*, page 215.

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 148.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read/Write

Reads from or writes to location.

Read

Reads from location.

Write

Writes to location.

Immediate breakpoints dialog box

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

Immediate

Trigger at: Edit...

Access Type

☒ Read

☐ Write

Action

Expression:

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 129. 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 148.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read

Reads from location.

Write

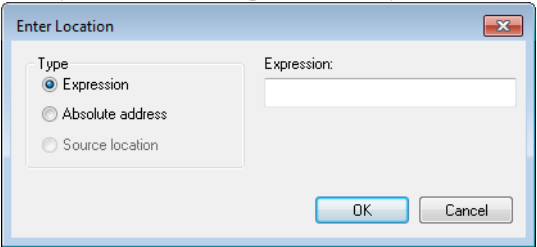
Writes to location.

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 132.

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 98.

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 153.

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.

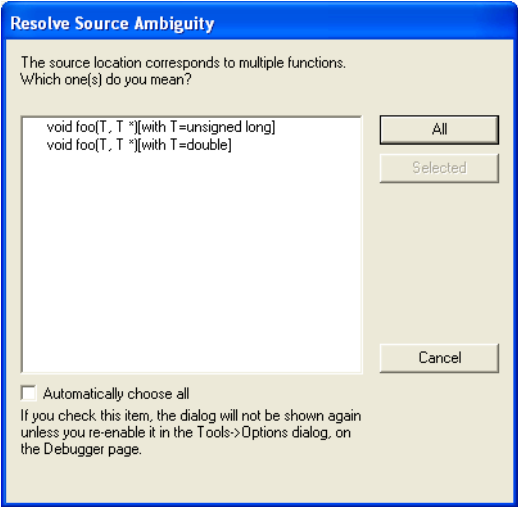
column specifies the column in which you want the breakpoint.

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 RX*.

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

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.

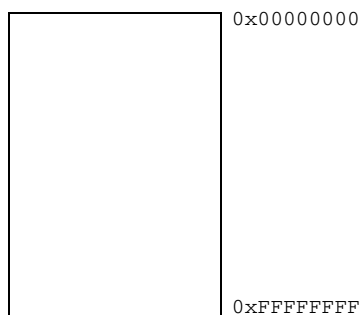
To view the memory contents for a specific variable, simply drag the variable to the **Memory** window or the **Symbolic Memory** window. The memory area where the variable is located will appear.



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 RX architecture has one zone, `Memory`, which covers the whole RX memory range.



Default zone `Memory`

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

Device-specific zones

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

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

If your hardware does not have the same memory layout as any of the predefined device description files, you must define customized zones in this file to be able to view the corresponding memory in the debugger.

For more information, see *Selecting a device description file*, page 47 and *Modifying a device description file*, page 53.

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 Access Setup dialog box*, page 183.

Monitoring memory and registers

These tasks are covered:

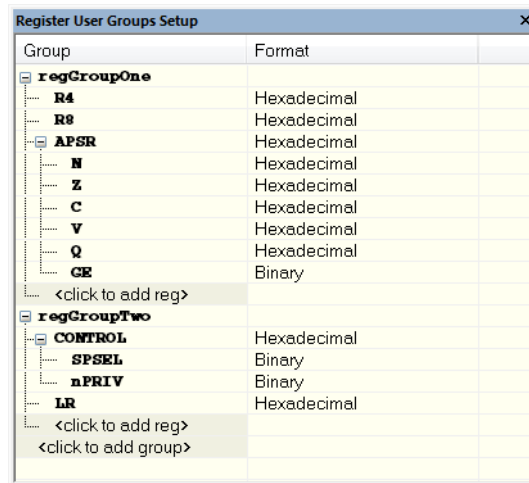
- Defining application-specific register groups
- Monitoring stack usage

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 176.

- 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 178.

MONITORING STACK USAGE

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

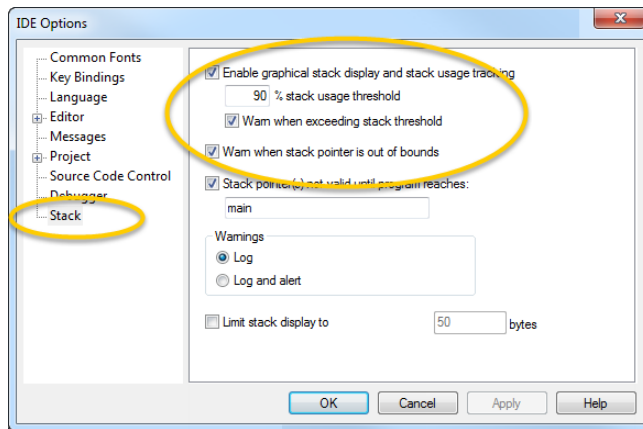
- Monitoring stack memory usage

- 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 RX*.

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.

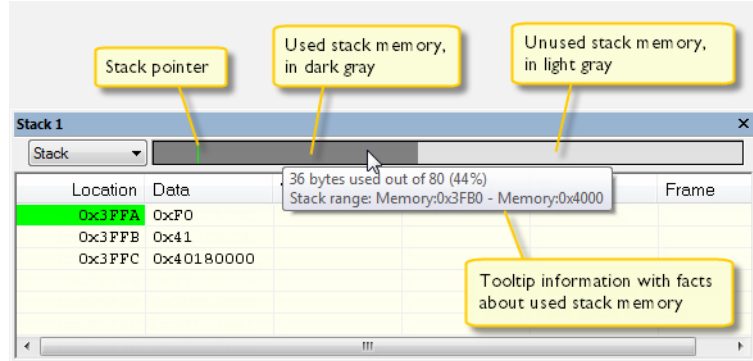
- 3 Choose **View>Stack>Stack 1** to open the **Stack** window.

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:

Current stack pointer

Stack 1					
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				

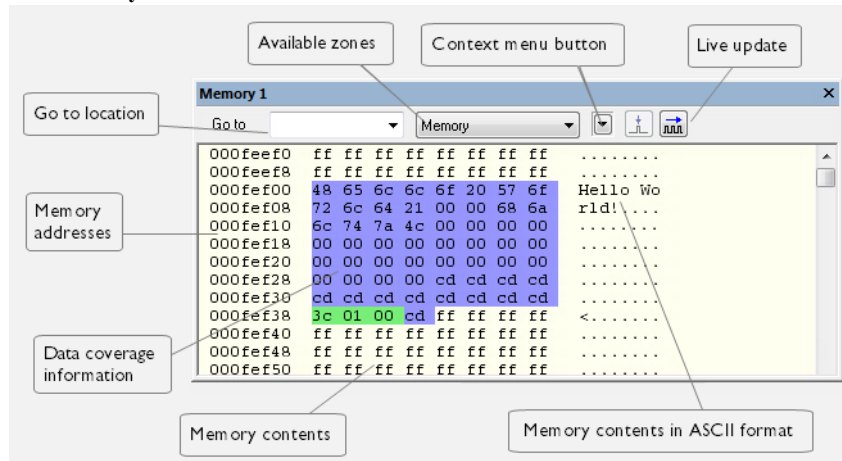
Reference information on memory and registers

Reference information about:

- *Memory window*, page 159
- *Memory Save dialog box*, page 163
- *Memory Restore dialog box*, page 164
- *Fill dialog box*, page 165
- *Symbolic Memory window*, page 166
- *Stack window*, page 169
- *Registers window*, page 173
- *Register User Groups Setup window*, page 176
- *SFR Setup window*, page 178
- *Edit SFR dialog box*, page 181
- *Memory Access Setup dialog box*, page 183
- *Edit Memory Access dialog box*, page 185

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 50.

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 153.

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.

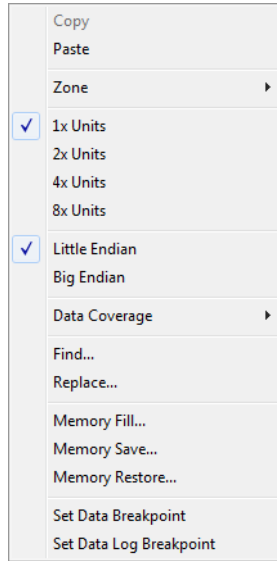
Data coverage is displayed with these colors:

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 and the E1/E20 driver.

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 153.

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 RX*.

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 RX*.

Memory Fill

Displays a dialog box, where you can fill a specified area with a value, see *Fill dialog box*, page 165.

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 163.

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 164.

Set Data Breakpoint

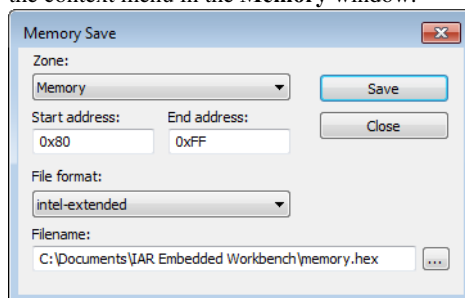
Sets breakpoints directly in the **Memory** window. The breakpoint is not highlighted—you can see, edit, and remove it in the breakpoint dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the Memory window*, page 131.

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 125 and *Getting started using data logging*, page 215.

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 153.

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

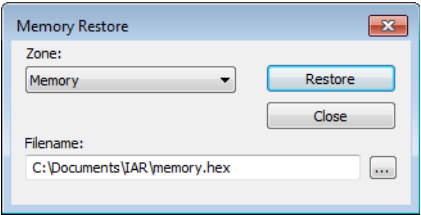
Specify the destination file to be used. A browse button is available.

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 153.

Filename

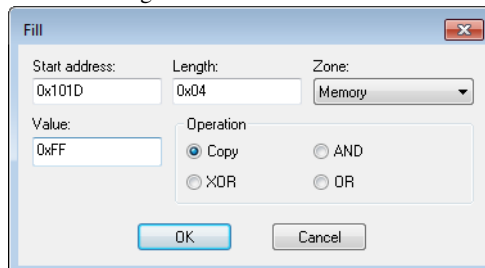
Specify the file to be read. A browse button is available.

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 153.

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

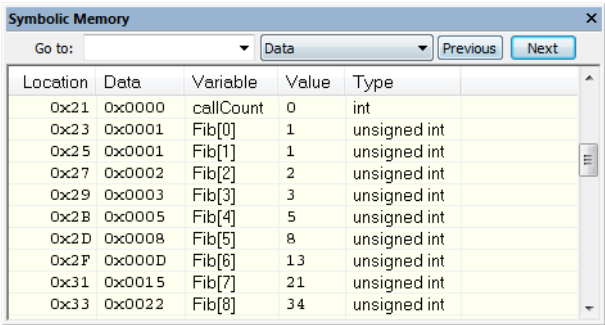
An **XOR** operation will be performed between Value and the existing contents of memory before writing the result to memory.

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 50.

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 153.

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.

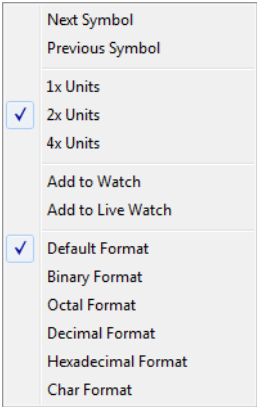
There are several different ways to navigate within the memory space:

- 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

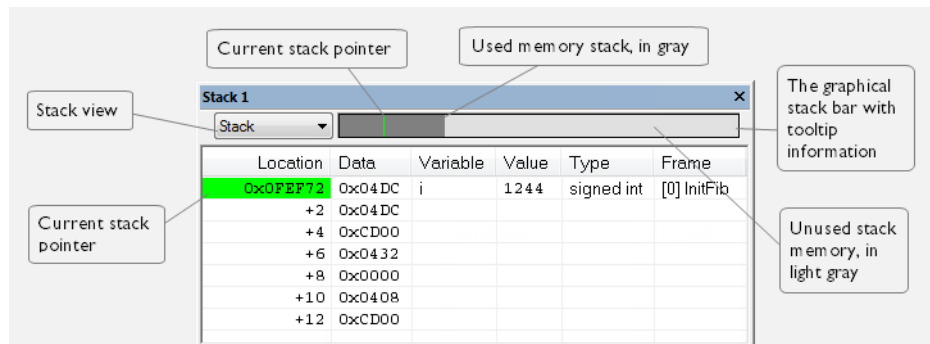
Displays the memory contents in hexadecimal format.

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 127.

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

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 microcontrollers with multiple stacks.

The graphical stack bar

Displays the state of the stack graphically.

The left end of the stack bar represents the bottom of the stack, in other words, the position of the stack pointer when the stack is empty. The right end represents the end of the memory 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

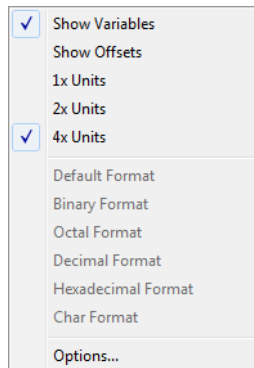
Displays the data type of the variable.

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:

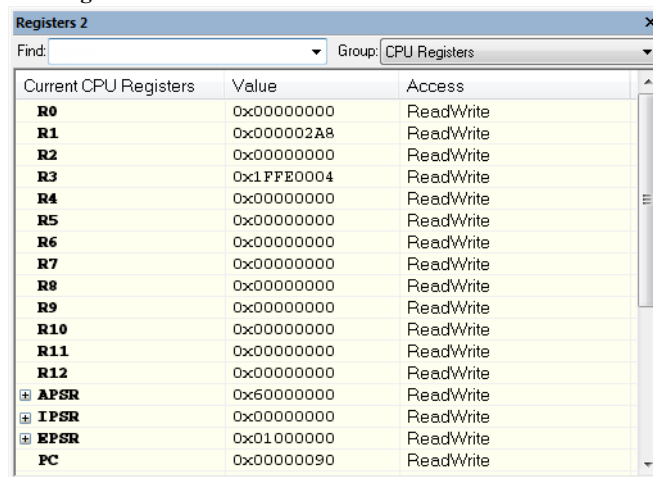
- | | |
|------------------|---|
| Variables | The display setting affects only the selected variable, not other variables. |
| Array elements | The display setting affects the complete array, that is, the same display format is used for each array element. |
| Structure fields | All elements with the 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 RX*.

Registers window

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



The screenshot shows a window titled 'Registers 2' with a 'Find:' search bar and a 'Group: CPU Registers' dropdown. Below is a table with three columns: 'Current CPU Registers', 'Value', and 'Access'.

Current CPU Registers	Value	Access
R0	0x00000000	ReadWrite
R1	0x000002A8	ReadWrite
R2	0x00000000	ReadWrite
R3	0x1FFE0004	ReadWrite
R4	0x00000000	ReadWrite
R5	0x00000000	ReadWrite
R6	0x00000000	ReadWrite
R7	0x00000000	ReadWrite
R8	0x00000000	ReadWrite
R9	0x00000000	ReadWrite
R10	0x00000000	ReadWrite
R11	0x00000000	ReadWrite
R12	0x00000000	ReadWrite
APSR	0x60000000	ReadWrite
IPSR	0x00000000	ReadWrite
EPSR	0x01000000	ReadWrite
PC	0x00000090	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 50.

To enable predefined register groups:

- 1 Select a device description file that suits your device, see *Selecting a device description file*, page 47. 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 154.

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 178.

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

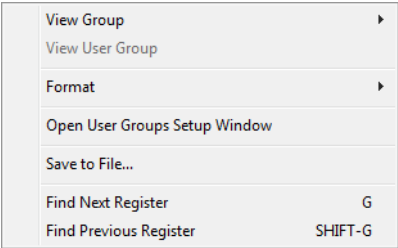
The access type of the register. Some of the registers are read-only, while others are write-only.

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 154.

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 176.

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

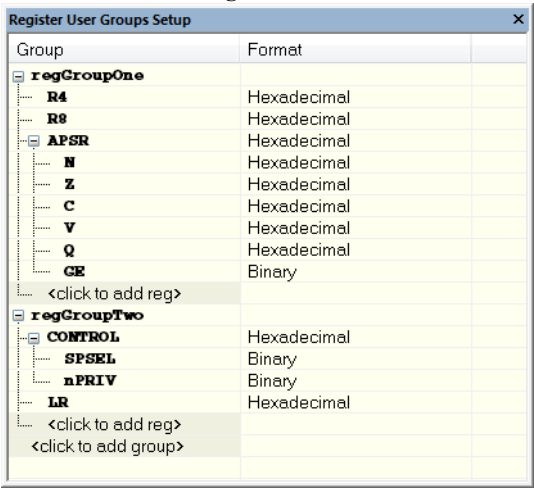
Finds the predefined register or register group that comes immediately after what your search found. After the last register was found, this search wraps around and finds the first register again.

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.

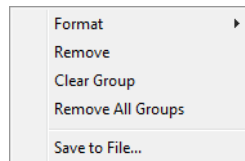
A dimmed register name indicates that it is not supported by the selected device.

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

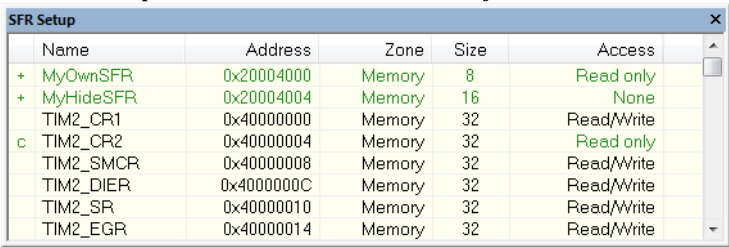
Deletes all user-defined register groups from your project.

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 181. For factory-defined SFRs (that is, retrieved from the `ddf` 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 153.

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.

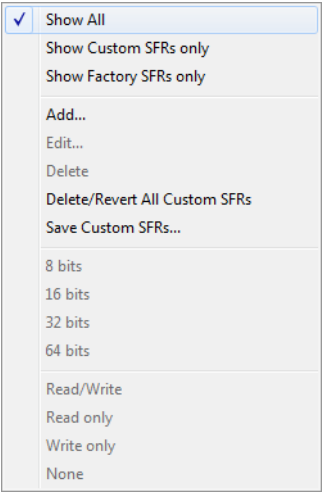
You can click a column header to sort the SFRs according to the column property.

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 181.

Edit

Displays the **Edit SFR** dialog box where you can edit an SFR, see *Edit SFR dialog box*, page 181.

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

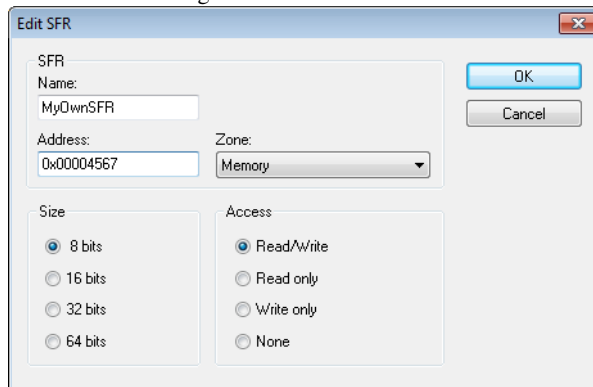
Selects display format for the selected SFR, which can be **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.

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 178.

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 0× 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 0×4567.

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

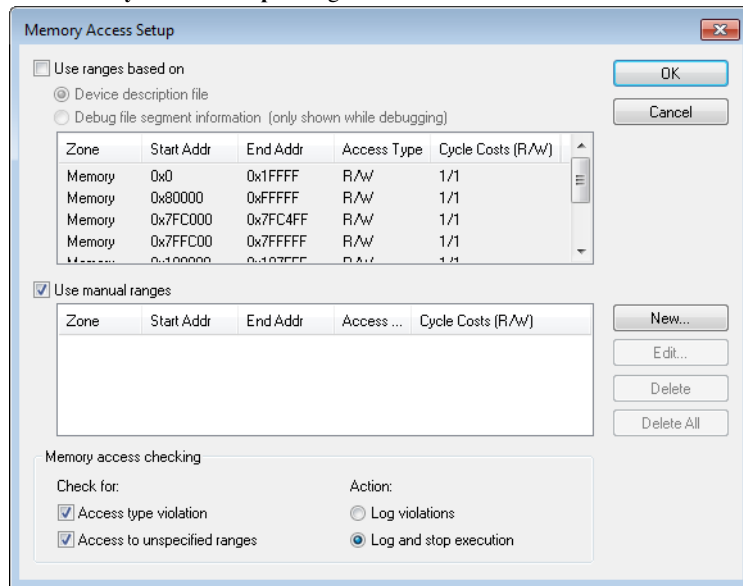
Selects the size of the SFR. Choose between **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.

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 Access Setup dialog box

The **Memory Access Setup** 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.

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

For information about the columns and the properties displayed, see *Edit Memory Access dialog box*, page 185. See also *Memory configuration for the C-SPY simulator*, page 153.

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 47.

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.

Use manual ranges

Specify your own ranges manually via the **Edit Memory Access** dialog box. To open this dialog box, click **New** to specify a new memory address range, or select an existing memory address range and choose **Edit** to modify it. For more information, see *Edit Memory Access dialog box*, page 185.

The ranges you define manually are saved between debug sessions.

Memory access checking

Check for determines what to check for:

- **Access type violation**
- **Access to unspecified ranges**

Action selects the action to be performed if an access violation occurs. Choose between:

- **Log violations**
- **Log and stop execution**

Any violations are logged in the **Debug Log** window.

Buttons

These buttons are available for manual ranges:

New

Opens the **Edit Memory Access** dialog box, where you can specify a new memory address range and associate an access type with it, see *Edit Memory Access dialog box*, page 185.

Edit

Opens the **Edit Memory Access** dialog box, where you can edit the selected memory address range. See *Edit Memory Access dialog box*, page 185.

Delete

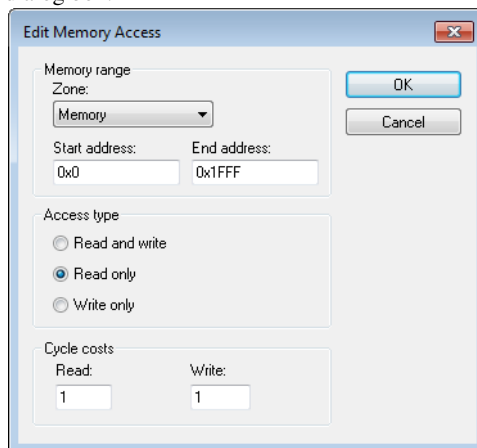
Deletes the selected memory address range definition.

Delete All

Deletes all defined memory address range definitions.

Edit Memory Access dialog box

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



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

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 153.

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 to the memory address range. Choose between:

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

Cycle costs

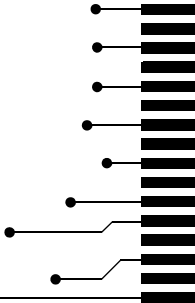
- **Read**
- **Write.**

The cycle cost can be specified individually for read and write accesses, because it can differ.

Part 2. Analyzing your application

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

- Trace
- The application timeline
- Profiling
- Analyzing code performance
- Code coverage
- Power debugging
- C-RUN runtime error checking





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 215
- *Power debugging*, page 269
- *Getting started using interrupt logging*, page 338
- *Profiling*, page 241

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.

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

Trace data is a continuously collected sequence of every executed instruction for a selected portion of the execution.

Trace features in C-SPY

In C-SPY, you can use the trace-related windows—**Trace**, **Function Trace**, **Timeline**, and **Find in Trace**.

Depending on your C-SPY driver, you:

- Can set various types of trace breakpoints to control the collection of trace data.
- Have access to windows such as the **Power Log**, **Interrupt Log**, **Interrupt Log Summary**, **Data Log**, and **Data Log Summary**.

In addition, several other features in C-SPY also use trace data, features such as Profiling, Code coverage, and Instruction profiling.

REQUIREMENTS FOR USING TRACE

The C-SPY simulator supports trace-related functionality, and there are no specific requirements.

To use trace in your hardware debugger systems, you need debug components (hardware, in some cases a debug probe, and a C-SPY driver) that all support trace. All C-SPY hardware debugger drivers support trace.

Note: The specific set of debug components you are using (hardware, a debug probe, and a C-SPY driver) determine which trace features in C-SPY that are supported.

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

- 1 For the C-SPY E20 emulator, before you start C-SPY you must choose **E1/E20 Emulator>Hardware Setup** and select **Trace** as the **Emulator mode**.

Note: If you are using the C-SPY simulator, the E1, E2, or E2 Lite or EZ-CUBE2 emulator or the J-Link debug probe, you do not need to perform this step.

- 2 Start C-SPY and choose **C-SPY driver>Trace Settings**. In the **Trace Settings** dialog box that is displayed, check if you need to change any of the default settings. For more information, see *Trace Settings dialog box*, page 193.

Note: If you are using the C-SPY simulator, just start C-SPY.



- 3 Open the **Trace** window—available from the driver-specific menu—and click the **Activate** button to enable collecting trace data.
- 4 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 196.

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 205 and *Trace Stop Trigger breakpoint dialog box*, page 206, respectively.

Note: Trace information can also be collected from specified memory locations by using data trace collection breakpoints, see *Data Trace Collection breakpoints dialog box*, page 207.

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:



- I 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 207.

- 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 209.

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.

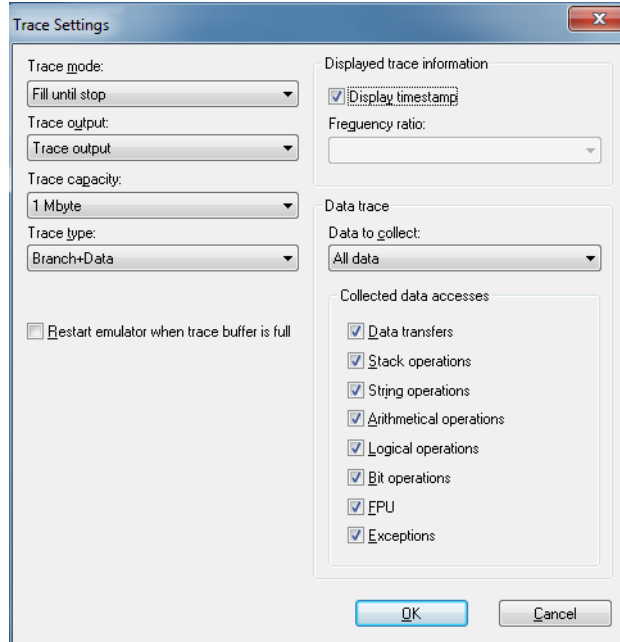
Reference information on trace

Reference information about:

- *Trace Settings dialog box*, page 193
- *Trace window*, page 196
- *Function Trace window*, page 203
- *Trace Start Trigger breakpoint dialog box*, page 205
- *Trace Stop Trigger breakpoint dialog box*, page 206
- *Data Trace Collection breakpoints dialog box*, page 207
- *Find in Trace dialog box*, page 207
- *Find in Trace window*, page 209

Trace Settings dialog box

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



Use this dialog box to configure trace generation and collection.

See also *Getting started with trace*, page 190.

Requirements

A C-SPY hardware debugger driver.

Trace mode

Selects the trace collection mode. Choose between:

Fill until stop

Continues to collect trace data until the execution stops or a trace stop breakpoint stops the collection.

Fill until full

Continues to collect trace data until the buffer is full.

Trace output

Controls the generation of trace data. Choose between:

CPU execution

CPU execution is given priority, meaning that some trace data might be lost.

Trace output

Generating trace data is given priority. Because the CPU execution is paused when trace data is generated, execution speed slows down.

Do not output

No trace data is generated. The trace buffer of the MCU will be used.

Note: For the E1, E2, and E2 Lite/EZ-CUBE2 emulators and the J-Link debug probe, only the setting Do not output is available.

Trace capacity

Sets the size of the trace buffer: 1, 2, 4, 8, 16, or 32 Mbytes.

Note: This option is only available for the E20 emulator.

Trace type

Selects the type of trace data that is collected. Choose between:

Branch

Collects source and destination address information on branches that occurred during program execution.

Branch+Data

Collects branch and data access information. Only available for the RX600 architecture.

Data

Collects data information on events that occurred during program execution. For the RX100 and RX200 architectures, only data accesses set up with data trace collection breakpoints can be traced.

Restart emulator when trace buffer is full

Halts execution when the trace buffer is full and restarts the emulator after the buffer has been read by the emulator. This option offers full trace, but might result in very large amounts of trace data.

Note: This option is only available for the E20 emulator using a 38-pin connection, and requires that the option **Trace output** is set to **CPU execution** or **Trace output**.

Display timestamp

Displays the timestamp of the collected trace data. This option is only available for the RX600 architecture.

Frequency ratio

Selects a frequency division ratio for the timestamp counter. This option is only available for the RX64M group of MCUs. Choose between:

No frequency division

The frequency of the timestamp counter is equal to the selected clock source frequency.

1/16 frequency

The frequency of the timestamp counter is 1/16 of the selected clock source frequency.

1/256 frequency

The frequency of the timestamp counter is 1/256 of the selected clock source frequency.

1/4096 frequency

The frequency of the timestamp counter is 1/4096 of the selected clock source frequency.

Data to collect

Selects the data trace information to collect. Choose between:

All data

Collects all data accesses.

Data trace collection breakpoints

Collects only the data accesses that are triggered by data trace collection breakpoints.

Collected data accesses

Selects which types of memory accesses to collect and display in the Trace window. Choose between:

Data transfers

Displays trace information collected from data transfers.

Stack operations

Displays trace information collected from stack operations.

String operations

Displays trace information collected from string operations.

Arithmetical operations

Displays trace information collected from arithmetical operations.

Logical operations

Displays trace information collected from logical operations.

Bit operations

Displays trace information collected from bit operations.

FPU

Displays trace information collected from FPU instructions.

Exceptions

Displays trace information collected from exceptions.

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 190.

Requirements

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

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 192.

**Find**

Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 207.

**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.

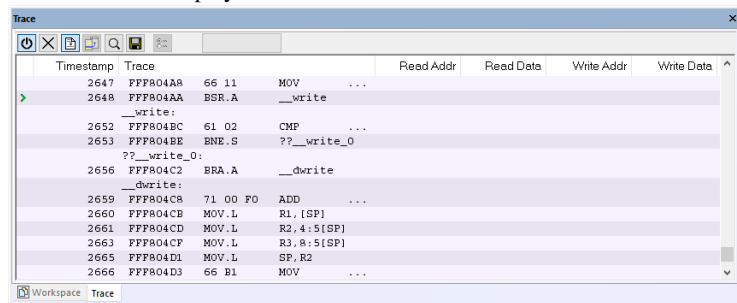
For the C-SPY hardware debugger drivers, this button displays the **Trace Settings** dialog box, see *Trace Settings dialog box*, page 193.

**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.








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.

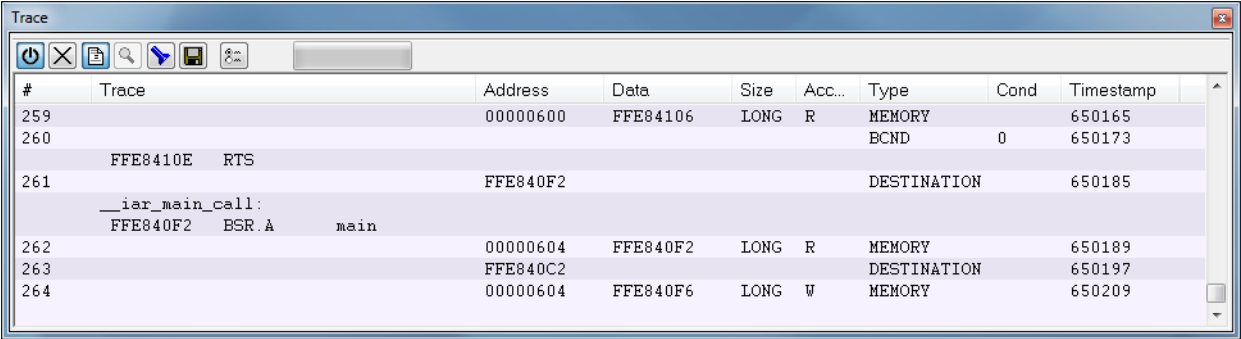
Read Addr, Read Data, Write Addr, Write Data

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 hardware debugger drivers)

This area displays a sequence of trace data collected from the hardware debugger system. In addition, the window can display the assembler source code for branch trace data.



#	Trace	Address	Data	Size	Acc...	Type	Cond	Timestamp
259		00000600	FFE84106	LONG	R	MEMORY		650165
260	FFE8410E RTS					BCND	0	650173
261		FFE840F2				DESTINATION		650185
	__iar_main_call: FFE840F2 BSR.A main							
262		00000604	FFE840F2	LONG	R	MEMORY		650189
263		FFE840C2				DESTINATION		650197
264		00000604	FFE840F6	LONG	W	MEMORY		650209

This area contains these columns for the C-SPY hardware debugger drivers:

#

The frame number for trace data from the hardware debugger system. Restarts from zero each time data is collected.

Trace

For branch trace, the assembler source code is displayed if this information could be collected.

Address

The memory address where the data access occurred.

Data

The data value that was read or written to the address.

Size

The size of the data read or written. This can be `byte`, `word` or `long`.

Access

The type of the memory access, `R` for read or `W` for write.

Type

The type of the collected data, one of:

- **Memory** – Data access
- **Destination** – The destination address of a branch
- **BCND** – Conditional branch information
- **Lost** – The corresponding trace information was lost
- **Source** – The address that was last executed before an interrupt (RX600 series) or where a branch occurred (RX100 and RX200 series)

Cond

1 indicates that the condition for executing a conditional branch instruction was satisfied; 0 indicates that it was not satisfied.

Each row can show information on up to 15 branches.

Timestamp

The timestamp of the collected data. This column is only available for the RX600 architecture. The timestamp counter frequency varies depending on the RX MCU you are using:

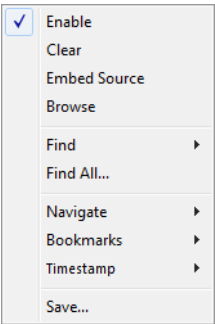
- For RX61x and RX62x MCUs when $\text{EXTAL} \times 8 \leq 100 \text{ MHz}$, the timestamp counter frequency is $\text{EXTAL} \times 8$.
- For RX61x and RX62x MCUs when $\text{EXTAL} \times 8 > 100 \text{ MHz}$, the timestamp counter frequency is $\text{EXTAL} \times 4$.
- For RX63x and RX64x MCUs using the EXTAL pin, the timestamp counter frequency is 1/2 the selected clock source frequency (or equal to the selected clock source frequency if the division ratio is set to 1:1 by the `SCKCR.ICK` bit).

- For RX63x and RX64x MCUs not using the EXTAL pin, the timestamp counter frequency is equal to the selected clock source frequency. For the RX64M MCUs, you can use the **Frequency ratio** option in the **Trace Settings** dialog box to decrease this frequency.

An italicized row indicates that the previous row and the italicized 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 192.

Find>Find (F)

Displays a dialog box where you can perform a search in the **Trace** window, see *Find in Trace dialog box*, page 207. 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 207. The search results are displayed in the **Find in Trace** window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 209.

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

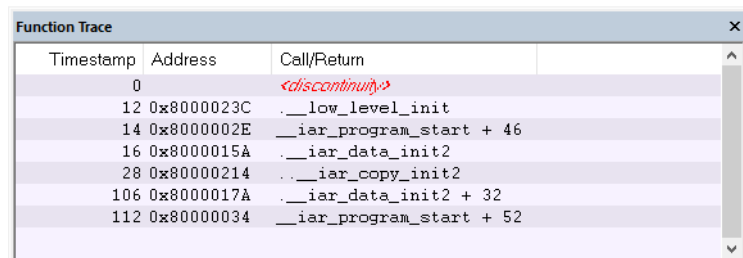
Removes the reference “zero” point from the trace data and restores the original timestamps of all rows.

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

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

Toolbar

For information about the toolbar, see *Trace window*, page 196.

Display area

There are two sets of columns available, and which set is used in your debugging system depends on the debug probe and which trace sources that are available:

- The available columns are the same as in the **Trace** window, see *Trace window*, page 196.
- For , these columns are available:

Timestamp

The number of cycles elapsed to this point according to the timestamp in the debug probe.

Address

The address of the executed instruction.

Call/Return

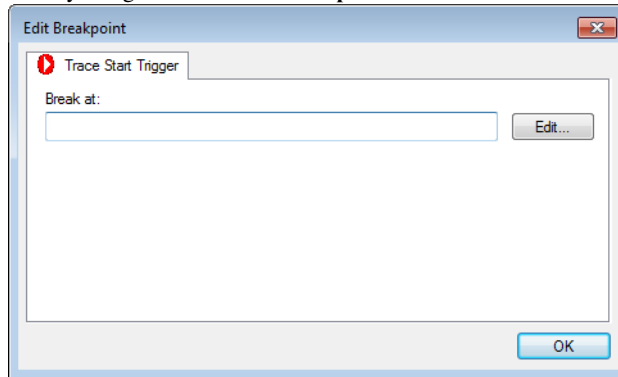
The function that was called or returned to.

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 196.

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 206 and *Trace data collection using breakpoints*, page 191.

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

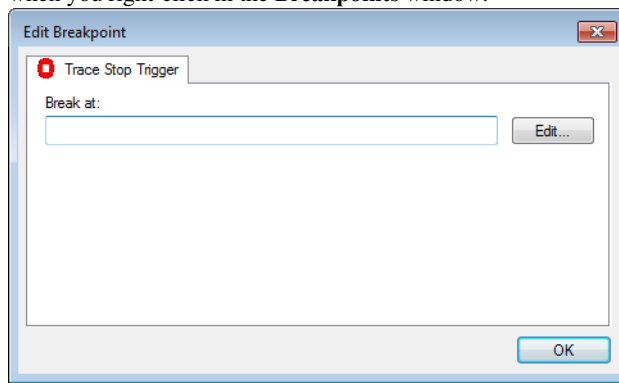
Can be used with all C-SPY debugger drivers and debug probes.

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 148.

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 205 and *Trace data collection using breakpoints*, page 191.

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

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 148.

Data Trace Collection breakpoints dialog box

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

Use this dialog box to set breakpoints to collect trace data from one of the hardware debugger drivers. Data trace collection breakpoints do not break the execution.

This dialog box is designed identically to the **Data** breakpoints dialog box, see *Data breakpoints dialog box (C-SPY hardware debugger drivers)*, page 144, for reference information.

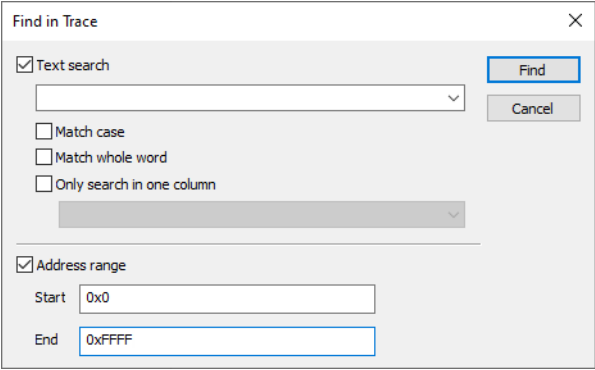
Requirements

A C-SPY hardware debugger driver.

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 209.

See also *Searching in trace data*, page 191.

Requirements

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

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

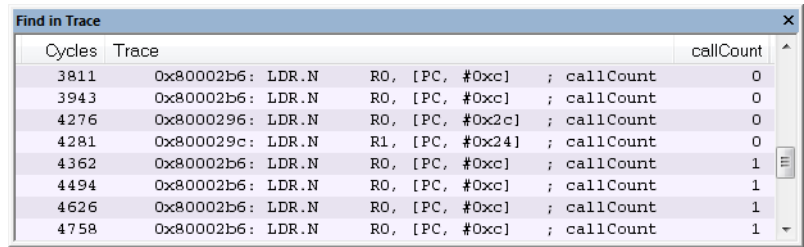
Searches only in the column you selected from the drop-down list.

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.



The screenshot shows the 'Find in Trace' window with a table of results. The table has three columns: 'Cycles', 'Trace', and 'callCount'. The data is as follows:

Cycles	Trace	callCount
3811	0x80002b6: LDR.N R0, [PC, #0xc] ; callCount	0
3943	0x80002b6: LDR.N R0, [PC, #0xc] ; callCount	0
4276	0x8000296: LDR.N R0, [PC, #0x2c] ; callCount	0
4281	0x800029c: LDR.N R1, [PC, #0x24] ; callCount	0
4362	0x80002b6: LDR.N R0, [PC, #0xc] ; callCount	1
4494	0x80002b6: LDR.N R0, [PC, #0xc] ; callCount	1
4626	0x80002b6: LDR.N R0, [PC, #0xc] ; callCount	1
4758	0x80002b6: LDR.N R0, [PC, #0xc] ; callCount	1

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 207.

See also *Searching in trace data*, page 191.

Requirements

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

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 189

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*. The graphs appear either in the **Timeline** window or the **Sampled graphs** window, depending on the source of the data
- As detailed logs
- As summaries of the logs.

Depending on the capabilities of your hardware, the debug probe, and the C-SPY driver you are using, 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 84 and *Function Trace window*, page 203, 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*.

Power logging Based on logged power measurement samples generated by the debug probe or associated hardware. Choose to display the power logs:

- In the **Timeline** window, as a graph of the power measurement samples.
- In the **Power Log** window.

Power logs can be useful for finding peaks in the power consumption and by double-clicking on a value you can see the corresponding source code. The precision depends on the frequency of the samples, but there is a good chance that you find the source code sequence that caused the peak.

For more information, see the chapter *Power debugging*.

REQUIREMENTS FOR TIMELINE SUPPORT

Depending on the capabilities of the hardware, the debug probe, and the C-SPY driver you are using, trace-based timeline information is supported for:

Target system	Call Stack graph	Data Log graph	Interrupt Log graph	Power Log graph
C-SPY simulator	Yes	Yes	Yes	—
C-SPY J-Link driver	—	—	—	Yes
C-SPY E2 driver	—	—	—	Yes

Table 10: Supported graphs in the Timeline window

For more information about requirements related to trace data, see *Requirements for using trace*, page 190.

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
- Getting started using data sampling

See also:

- *Debugging in the power domain*, page 275
- *Using the interrupt system*, page 335

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 211.

- 1 Choose **J-Link>Operating Frequency** and specify the operating frequency of the MCU.

If you are using the C-SPY simulator, choose **Simulator>Simulated Frequency** to set up a frequency that matches the simulated hardware.

- 2 Choose **Timeline** from the C-SPY driver menu to open the **Timeline** window.

- 3 In the **Timeline** window, right-click in the window and choose **Select Graphs** from the context menu to select which graphs to be displayed.
- 4 In the **Timeline** window, right-click in the graph area and choose **Enable** from the context menu to enable a specific graph.
- 5 For the Data Log graph, you must set a Data Log breakpoint for each variable you want a graphical representation of in the **Timeline** window. See *Data Log breakpoints dialog box*, page 146.
- 6 Click **Go** on the toolbar to start executing your application. The graphs that you have enabled appear.

NAVIGATING IN THE GRAPHS

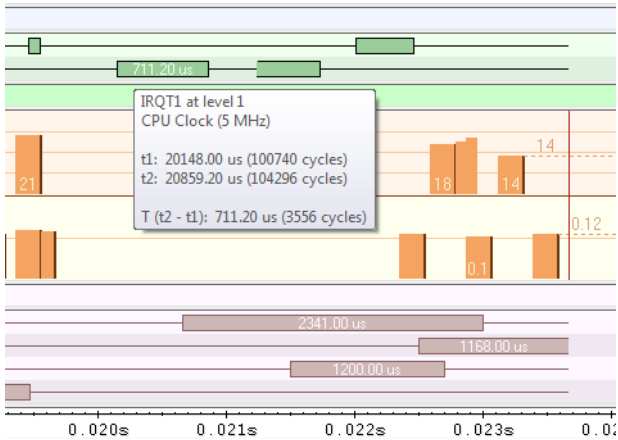
After you have performed the steps in *Displaying a graph in the Timeline window*, page 213, you can use any of these alternatives to navigate in the graph:

- 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

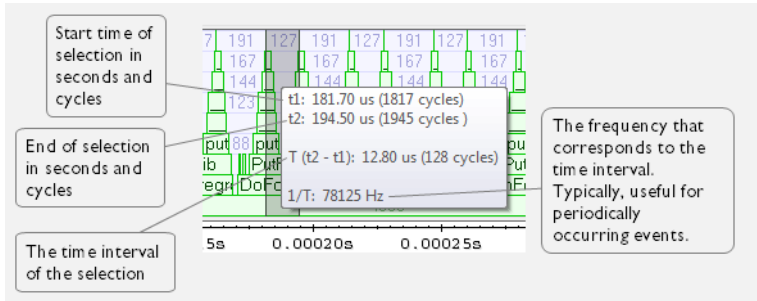
The **Timeline** window provides a set of tools for analyzing the graph data.

- 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 213.
- 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

- I 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 125.

- 2 Choose **C-SPY driver>Data Log** to open the **Data Log** window. Optionally, you can also choose:
 - **C-SPY driver>Data Log Summary** to open the **Data Log Summary** window
 - **C-SPY driver>Timeline** to open the **Timeline** window to view the Data Log graph.
- 3 From the context menu, available in the **Data Log** window, choose **Enable** to enable the logging.
- 4 Start executing your application program to collect the log information.
- 5 To view the data log information, look in the **Data Log** window, the **Data Log Summary** window, or the Data Log graph in the **Timeline** window.
- 6 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
- 7 To disable data logging, choose **Disable** from the context menu in each window where you have enabled it.

GETTING STARTED USING DATA SAMPLING

- 1 Choose **C-SPY driver>Data Sample Setup** to open the **Data Sample Setup** window.
- 2 In the **Data Sample Setup** window, perform these actions:
 - In the **Expression** column, type the name of the variable for which you want to sample data. The variable must be an integral type with a maximum size of 32 bits and you can specify up to four variables. Make sure that the checkbox is selected for the variable that you want to sample.
 - In the **Sampling interval** column, type the number of milliseconds to pass between the samples.
- 3 To view the result of data sampling, you must enable it in the window in question:
 - Choose **C-SPY driver>Data Sample** to open the **Data Sample** window. From the context menu, choose **Enable**.

- Choose **C-SPY driver>Sampled Graphs** to open the **Sampled Graphs** window. From the context menu, choose **Enable**.
- 4** Start executing your application program. This starts the data sampling. When the execution stops, for example because a breakpoint is triggered, you can view the result either in the **Data Sample** window or as the Data Sample graph in the **Sampled Graphs** 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 data sampling, choose **Disable** from the context menu in each window where you have enabled it.

Reference information on application timeline

Reference information about:

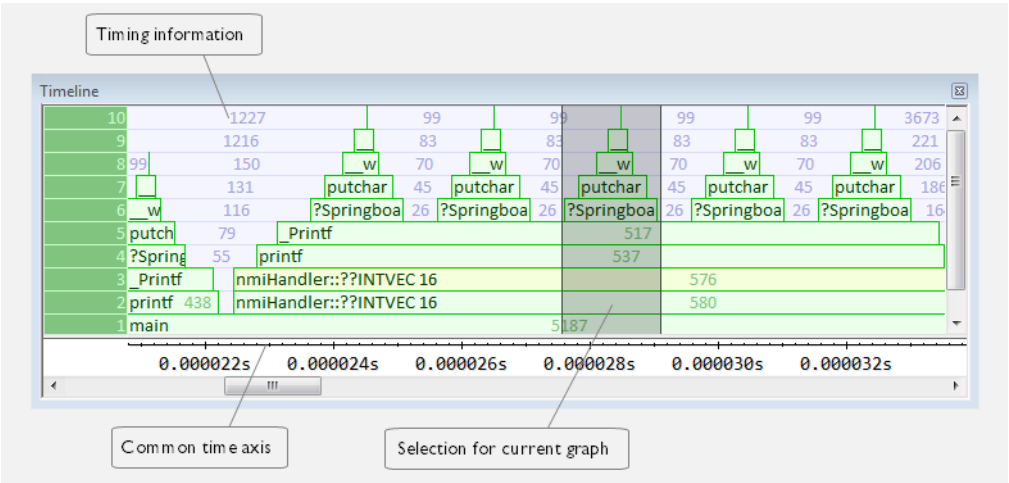
- *Timeline window—Call Stack graph*, page 218
- *Timeline window—Data Log graph*, page 221
- *Data Log window*, page 225
- *Data Log Summary window*, page 228
- *Data Sample window*, page 230
- *Data Sample Setup window*, page 232
- *Sampled Graphs window*, page 234
- *Viewing Range dialog box*, page 238

See also:

- *Timeline window—Interrupt Log graph*, page 350
- *Timeline window—Power graph*, page 285

Timeline window—Call Stack graph

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



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

The C-SPY simulator.

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.

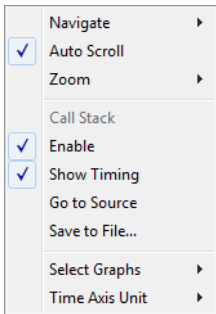
At the bottom of the window, there is a shared time axis that uses seconds or cycles as the time unit.

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.

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**.

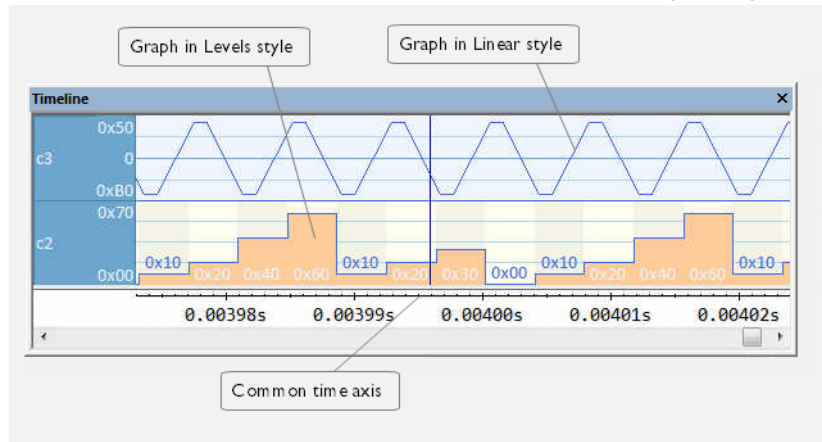
If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.

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. See *Selecting a time interval for profiling information*, page 247.

Timeline window—Data Log graph

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



This window displays trace data represented as different graphs, in relation to a shared time axis.

The Data Log graph displays the data logs collected by the trace system, for up to four different variables or address ranges specified as Data Log breakpoints.

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 for the Data Log graph

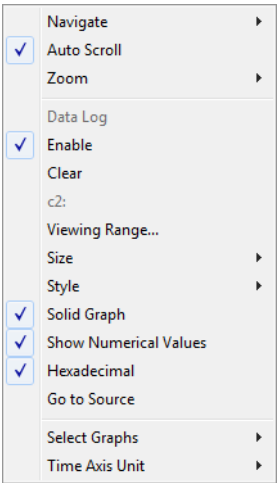
Where:

- The label area at the left end of the graph displays the variable name or the address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the Y-axis for a variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the **Data Log** window, see *Data Log window*, page 225.
- The graph can be displayed either as a thin line between consecutive logs or as a rectangle for every log (optionally color-filled).
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system. A red question mark indicates a log without a value.

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

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.

Data Log

A heading that shows that the Data 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.

Variable

The name of the variable for which the Data Log-specific commands below apply. This menu command is context-sensitive, which means it reflects the Data Log graph you selected in the **Timeline** window (one of up to four).

Viewing Range

Displays a dialog box, see *Viewing Range dialog box*, page 238.

Size

Determines the vertical size of the graph—choose between **Small**, **Medium**, and **Large**.

Solid Graph

Displays the graph as a color-filled solid graph instead of as a thin line.

Show Numerical Value

Shows the numerical value of the variable, in addition to the graph.

Hexadecimal

Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.

Go To Source

Displays the corresponding source code in an editor window, if applicable.

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.

Data Log window

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

Time	Program Counter	I1	Address	s2	Address
0.160us	----			W 0x0000	@ 0x2004
0.160us	0xFFE00049	-	@ 0x2000		
24.480us	0xFFE000B5			R 0x0000	@ 0x2006
24.720us	0xFFE000BF			W 0x0042	@ 0x2004
24.760us	0xFFE000C6			R 0x0042	@ 0x2006
24.960us	0xFFE000E4	W 0x00004444	@ 0x2000		
78.760us	0xFFE00104			R 0x0042	@ 0x2004+?
79.000us	----			W 0x0084	@ 0x2004
100.800us	0xFFE00104			R 0x0084	@ 0x2006
101.040us	0xFFE0010E			W 0x00C6	@ 0x2004
136.840us	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 215.

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

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 125.

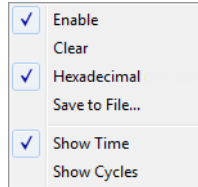
Address

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 215.

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 125.

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

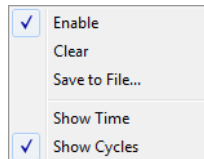
The information displayed depends on the C-SPY driver you are using.

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:



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

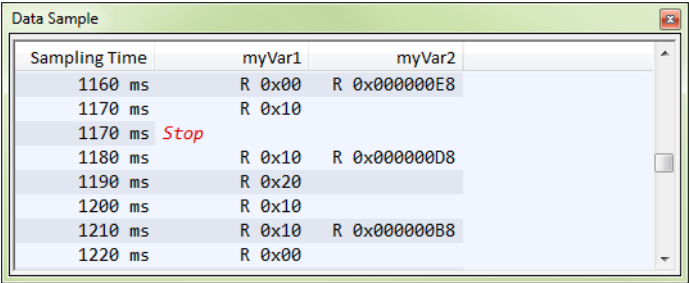
Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.

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.

Data Sample window

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



Sampling Time	myVar1	myVar2
1160 ms	R 0x00	R 0x000000E8
1170 ms	R 0x10	
1170 ms <i>Stop</i>		
1180 ms	R 0x10	R 0x000000D8
1190 ms	R 0x20	
1200 ms	R 0x10	
1210 ms	R 0x10	R 0x000000B8
1220 ms	R 0x00	

Use this window to view the result of the data sampling for the variables you have selected in the **Data Sample Setup** window.

Choose **Enable** from the context menu to enable data sampling.

See also *Getting started using data sampling*, page 216.

Requirements

Can be used with any supported hardware debugger system.

Display area

This area contains these columns:

Sampling Time

The time when the data sample was collected. Time starts at zero after a reset. Every time the execution stops, a red `stop` indicates when the stop occurred.

The selected expression

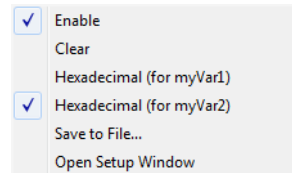
The column headers display the names of the variables that you selected in the **Data Sample Setup** window. The column cells display the sampling values for the variable.

There can be up to four columns of this type, one for each selected variable.

* You can double-click a row in the display area. If you have enabled the data sample graph in the **Sampled Graphs** window, the selection line will be moved to reflect the time of the row you double-clicked.

Context menu

This context menu is available:



These commands are available:

Enable

Enables data sampling.

Clear

Clears the sampled data.

Hexadecimal

Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.

Save to File

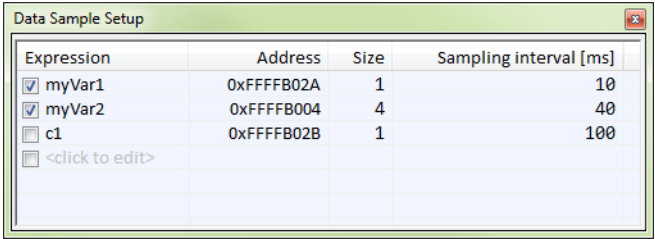
Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by TAB and LF characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.

Open setup window

Opens the **Data Sample Setup** window.

Data Sample Setup window

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



Use this window to specify up to four variables to sample data for. You can view the sampled data for the variables either in the **Data Sample** window or as graphs in the **Sampled Graphs** window.

See also *Getting started using data sampling*, page 216.

Requirements

Can be used with any supported hardware debugger system.

Display area

This area contains these columns:

Expression

Type the name of the variable which must be an integral type with a maximum size of 32 bits. Click the check box to enable or disable data sampling for the variable.

Alternatively, drag an expression from the editor window and drop it in the display area.

Variables in the expressions must be statically located, for example global variables.

Address

The actual memory address that is accessed. The column cells cannot be edited.

Size

The size of the variable, either 1, 2, or 4 bytes. The column cells cannot be edited.

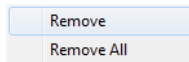
Sampling interval [ms]

Type the number of milliseconds to pass between the samples. The shortest allowed interval is 10 ms and the interval you specify must be a multiple of that.

Note that the sampling time you specify is just the interval (according to the Microsoft Windows calculations) for how often C-SPY checks with the C-SPY driver (which in turn must check with the MCU for a value). If this takes longer than the sampling interval you have specified, the next sampling will be omitted. If this occurs, you might want to consider increasing the sampling time.

Context menu

This context menu is available:



These commands are available:

Remove

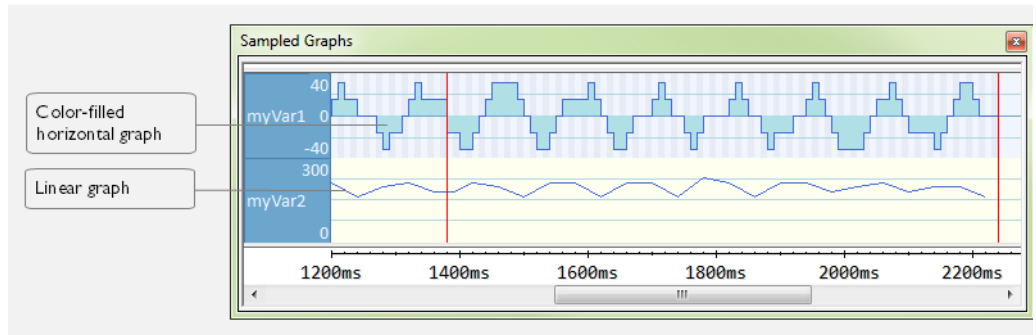
Removes the selected variable.

Remove All

Removes all variables.

Sampled Graphs window

The **Sampled Graphs** window is available from the C-SPY driver menu.



Use this window to display graphs for up to four different variables, and where:

- The graph displays how the value of the variable changes over time. The area on the left displays the limits, or range, of the Y-axis for the variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the **Data Sample** window, see *Data Sample window*, page 230.
- The graph can be displayed as levels, where a horizontal line—optionally color-filled—shows the value until the next sample. Alternatively, the graph can be linear, where a line connects consecutive samples.
- A red vertical line indicates the time of application execution stops.

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

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

- Right-click and choose **Zoom In** or **Zoom Out** from the context menu. Alternatively, use the + and – keys to zoom.
- Right-click in the graph and choose **Navigate** and the appropriate command to move backward and forward on the graph. Alternatively, use any of the shortcut keys: arrow keys, Home, End, and Ctrl+End.
- Double-click on a sample 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. Press Enter or right-click and choose **Zoom>Zoom to Selection** from the context menu. The selection zooms in.



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

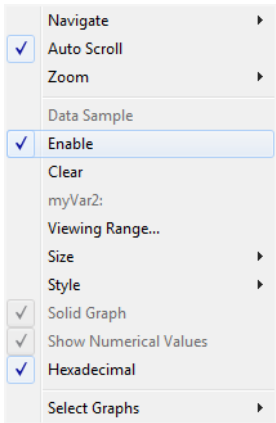
See also *Getting started using data sampling*, page 216.

Requirements

Can be used with any supported hardware debugger system.

Context menu

This context menu is available:



These commands are available:

Navigate

Commands for navigating in the graphs. Choose between:

Next moves the selection to the next relevant point in the graph. Shortcut key: right arrow.

Previous moves the selection 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: -

1us, 10us, 100us makes an interval of 1 microseconds, 10 microseconds, or 100 microseconds, respectively, fit the window.

1ms, 10ms, 100ms makes an interval of 1 millisecond, 10 milliseconds, or 100 milliseconds, respectively, fit the window.

1s, 10s, 100s makes an interval of 1 second, 10 seconds, or 100 seconds, respectively, fit the window.

1k s, 10k s, 100k s makes an interval of 1,000 seconds, 10,000 seconds, or 100,000 seconds, respectively, fit the window.

1M s, 10M s, makes an interval of 1,000,000 seconds or 10,000,000 seconds, respectively, fit the window.

Data Sample

A menu item that shows that the Data Sample-specific commands below are available.

Open Setup window (Data Sample Graph)

Opens the **Data Sample Setup** window.

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

Clears the sampled data.

Variable

The name of the variable for which the Data Sample-specific commands below apply. This menu item is context-sensitive, which means it reflects the Data Sample graph you selected in the **Sampled Graphs** window (one of up to four).

Viewing Range

Displays a dialog box, see *Viewing Range dialog box*, page 238.

Size

Controls the vertical size of the graph—choose between **Small**, **Medium**, and **Large**.

Style

Choose how to display the graph. Choose between:

Levels, where a horizontal line—optionally color-filled—shows the value until the next sample.

Linear, where a line connects consecutive samples.

Solid Graph

Displays the graph as a color-filled solid graph instead of as a thin line. This is only possible if the graph is displayed as Levels.

Hexadecimal

Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.

Show Numerical Value

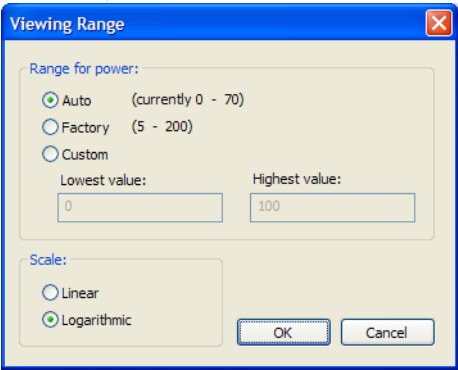
Shows the numerical value of the variable, in addition to the graph.

Select Graphs

Selects which graphs to display in the **Sampled Graphs** window.

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

One of these alternatives:

- The C-SPY simulator
- J-Link driver (J-Link debug probe only—not built-in J-Link)

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 RX*.

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

Instruction profiling can help you fine-tune your code on a very detailed level, especially for assembler source code. Instruction profiling can also help you to understand where your compiled C/C++ source code spends most of its time, and perhaps give insight into how to rewrite it for better performance.

BRIEFLY ABOUT THE PROFILER

Function profiling information is displayed in the **Function Profiler** window, that is, timing information for the functions in an application. Profiling must be turned on explicitly using a button on the window's toolbar, and will stay enabled until it is turned off.

Instruction profiling information is displayed in the **Disassembly** window, that is, the number of times each instruction has been executed.

Profiling sources

The profiler can use different mechanisms, or *sources*, to collect profiling information. Depending on the available trace source features, one or more of the sources can be used for profiling:

- *Trace (calls)*

The full instruction trace is analyzed to determine all function calls and returns. When the collected instruction sequence is incomplete or discontinuous, the profiling information is less accurate.

- *Trace (flat)/Sampling*

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.

Trace data for the E1, E2, and E2 Lite emulators and the J-Link debug probe is very limited. If you are using an E1, E2, E2 Lite or EZ-CUBE2 emulator or a J-Link debug probe, PC sampling provides much better profiling data than any other source.

Power sampling

Some debug probes support sampling of the power consumption of the development board, or components on the board. Each sample is associated with a PC sample and represents the power consumption (actually, the electrical current) for a small time interval preceding the time of the sample. When the profiler is set to use *Power Sampling*, additional columns are displayed in the **Profiler** window. Each power sample is associated with a function or code fragment, just as with regular PC Sampling.

Note that this does not imply that all the energy corresponding to a sample can be attributed to that function or code fragment. The time scales of power samples and instruction execution are vastly different—during one power measurement, the CPU has typically executed many thousands of instructions. Power Sampling is a statistics tool.

REQUIREMENTS FOR USING THE PROFILER

The C-SPY simulator supports the profiler; there are no specific requirements.

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

- A target board with built-in J-Link or a J-Link/J-Link Ultra debug probe and a J-Link RX adapter

- An E1, E2, E20, E2 Lite or an EZ-CUBE2 emulator.

This table lists the C-SPY driver profiling support:

C-SPY driver	Trace (calls)	Trace (flat)	Sampling	Power
C-SPY simulator	Yes	Yes	—	—
C-SPY E1/E20	—	Yes	Yes	—
C-SPY E2	—	Yes	Yes	Yes
C-SPY E2 Lite	—	Yes	Yes	—
C-SPY EZ-CUBE2	—	Yes	Yes	—
C-SPY J-Link	—	Yes	Yes	Yes

Table 11: 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
- Selecting a time interval for profiling information

GETTING STARTED USING THE PROFILER ON FUNCTION LEVEL

To display function profiling information in the Function Profiler window:

- 1 Build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Include debug information in output

Table 12: Project options for enabling the profiler

- 2 To set up the profiler for function profiling:

- If you use an E20 emulator and want to use another profiling source than PC sampling, you must choose **E1/E20 Emulator>Hardware Setup** and choose **Trace** as the **Emulator mode**.
- If you use the C-SPY simulator, an E1, E2, E2 Lite, or EZ-CUBE2 emulator or a J-Link debug probe, no specific settings are required.



- 3 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.
- 4 Start executing your application to collect the profiling information.
- 5 Profiling information is displayed in the **Function Profiler** window. To sort, click on the relevant column header.



- 6 When you start a new sampling, you can click the **Clear** button—alternatively, use the context menu—to clear the data.

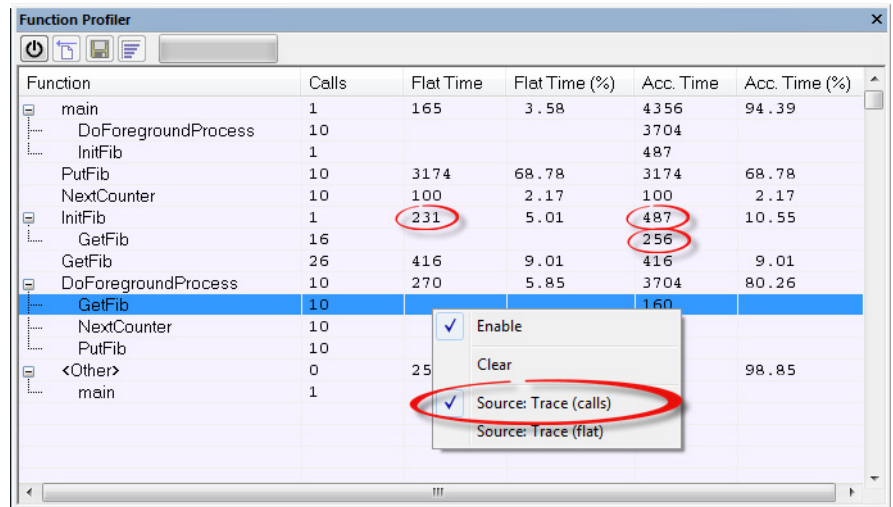
ANALYZING THE PROFILING DATA

Here follow some examples of how to analyze the data.

The first figure shows the result of profiling using **Source: Trace (calls)**. The profiler follows the program flow and detects function entries and exits.

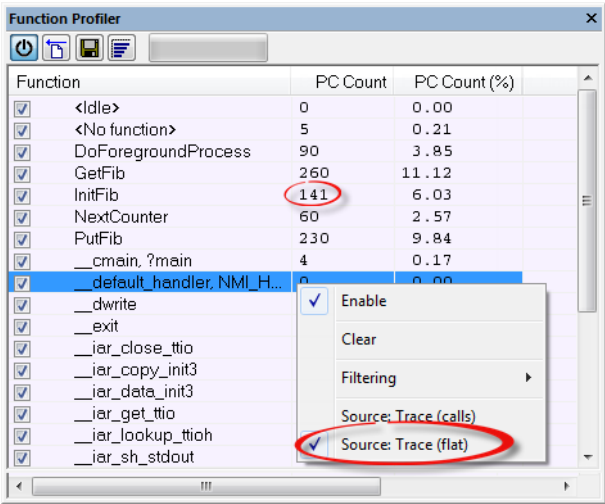
- For the **InitFib** function, **Flat Time** 231 is the time spent inside the function itself.
- For the **InitFib** function, **Acc Time** 487 is the time spent inside the function itself, including all functions **InitFib** calls.
- For the **InitFib/GetFib** function, **Acc Time** 256 is the time spent inside **GetFib** (but only when called from **InitFib**), including any functions **GetFib** calls.

- Further down in the data, you can find the **GetFib** function separately and see all of its subfunctions (in this case none).



The 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.



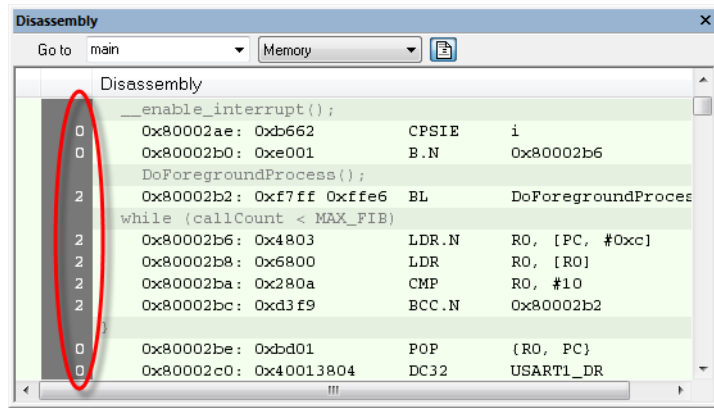
To secure valid data when using a debug probe, make sure to use the maximum trace buffer size and set a breakpoint in your code to stop the execution before the buffer is full.

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.

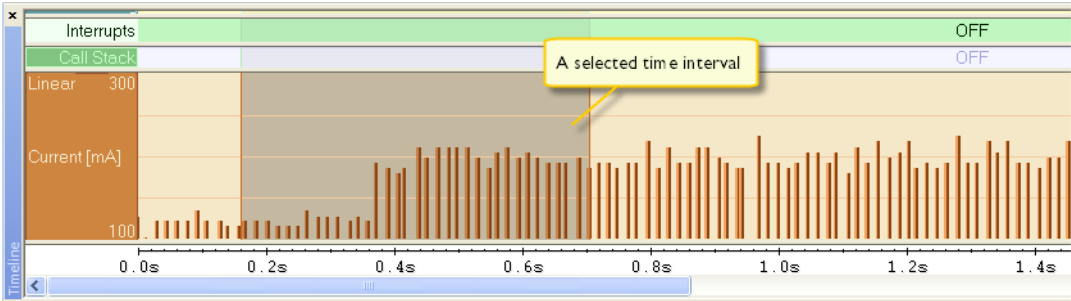
Instruction profiling attempts to use the same source as the function profiler. If the function profiler is not on, the instruction profiler will try to use first trace and then PC sampling as source. You can change the source to be used from the context menu that is available in the **Function Profiler**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

SELECTING A TIME INTERVAL FOR PROFILING INFORMATION

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

To select a time interval:

- 1 Choose **Function Profiler** from the C-SPY driver menu.
- 2 In the **Function Profiler** window, right-click and choose **Source: Sampling** from the context menu.
- 3 Execute your application to collect samples.
- 4 Choose **C-SPY driver>Timeline**.
- 5 In the **Timeline** window, click and drag to select a time interval.



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

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

160000.000us - 704000.000us					
Function	PC Count	PC Count (%)	Power Samples	Energy (%)	Avg
GetButtons()	791	33.10	9	30.82	198
Dly100us(void *)	463	19.37	7	15.38	127
GLCD_SPI_TranserByte(Int3...	353	14.77	4	8.32	120
memcpy	325	13.60	4	14.64	212
main()	288	12.05	6	20.07	193
GLCD_Backlight(Int8U)	108	4.52	2	6.77	196
GLCD_SendCmd(GLCD_Cm...	43	1.80	0	0.00	-
GLCD_SPI_SendBlock(plnt8...	19	0.79	2	4.00	116
GLCD_SetWindow(Int32U, Int...	0	0.00	0	0.00	-
GLCD_SetReset(Boolean)	0	0.00	0	0.00	-

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

Reference information on the profiler

Reference information about:

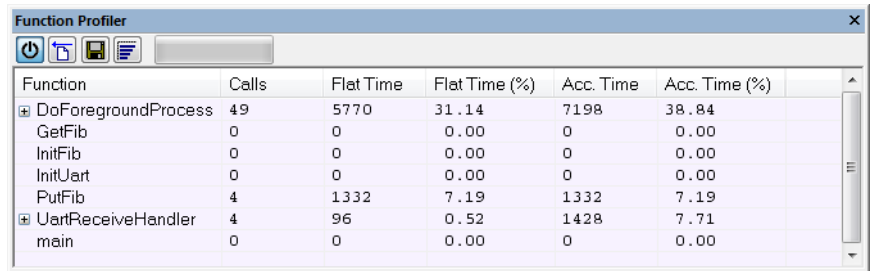
- *Function Profiler* window, page 249

See also:

- *Disassembly* window, page 79
- *Trace Settings* dialog box, page 193

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 (%)
<input checked="" type="checkbox"/> 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
<input checked="" type="checkbox"/> 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) or Sampling is selected, a checkbox appears on each line in the left-side margin of the window. Use these checkboxes to include or exclude lines from the profiling. Excluded lines are dimmed but not removed.

See also *Using the profiler*, page 243.

Requirements

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

Toolbar

The toolbar contains:



Enable/Disable

Enables or disables the profiler.



Clear

Clears all profiling data.



Save

Opens a standard **Save As** dialog box where you can save the contents of the window to a file, with tab-separated columns. Only non-expanded rows are included in the list file.



Graphical view

Overlays the values in the percentage columns with a graphical bar.

Progress bar

Displays a backlog of profiling data that is still being processed. If the rate of incoming data is higher than the rate of the profiler processing the data, a backlog is accumulated. The progress bar indicates that the profiler is still processing data, but also approximately how far the profiler has come in the process.

Note that because the profiler consumes data at a certain rate and the target system supplies data at another rate, the amount of data remaining to be processed can both increase and decrease. The progress bar can grow and shrink accordingly.

**Time-interval mode**

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

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 242.

Status field

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

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 242.

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 242.

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

Function (All sources)

The name of the profiled C function.

For Sampling source, sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels, is displayed.

Calls (Trace (calls))

The number of times the function has been called.

Flat time (Trace (calls))

The time expressed as the 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 number of executed instructions spent inside the function and everything called by the function.

Acc. time (%) (Trace (calls))

Accumulated time expressed as a percentage of the total time.

PC Count (Trace (flat) and Sampling)

The number of executed instructions (Trace) or PC samples (Sampling) associated with the function.

PC Count (%) (Trace (flat) and Sampling)

The number of executed instructions (Trace) or PC samples (Sampling) associated with the function as a percentage of the total number of executed instructions /PC samples.

Power Samples (Power Sampling)

The number of power samples associated with that function.

Energy (%) (Power Sampling)

The accumulated value of all measurements associated with that function, expressed as a percentage of all measurements.

Avg Current [mA] (Power Sampling)

The average measured value for all samples associated with that function.

Min Current [mA] (Power Sampling)

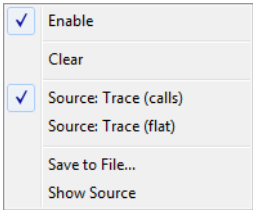
The minimum measured value for all samples associated with that function.

Max Current [mA] (Power Sampling)

The maximum measured value for all samples associated with that function.

Context menu

This context menu is available:



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 one of the modes Trace (flat) or Sampling.

Source

Selects which source to be used for the profiling information. See also *Profiling sources*, page 242.

Note that the available sources depend on the C-SPY driver you are using.

Choose between:

Sampling—the instruction count for instruction profiling represents the number of samples for each instruction.

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.

Power Sampling

Toggles power sampling information on or off.

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.

Analyzing code performance

- Introduction to performance analysis
- Analyzing performance
- Reference information on performance analysis.

Introduction to performance analysis

These topics are covered:

- Reasons for using performance analysis
- Briefly about performance analysis
- Requirements for performance analysis.

REASONS FOR USING PERFORMANCE ANALYSIS

The performance analyzing facility of the hardware debugger can measure a number of execution aspects to help you understand how well your application performs on the MCU.

Because performance analysis uses the debugger's performance measurement circuit to measure the execution time, it does not slow down the execution of your application.

BRIEFLY ABOUT PERFORMANCE ANALYSIS

The performance analysis is capable of measuring these execution aspects:

- the total time the execution takes
- the total number of cycles the execution takes
- the number of cycles spent processing interrupts and other exceptions
- the number of executed instructions
- the number of accepted interrupts and other exceptions.

The analysis can cover either the entire execution or execution between two breakpoints.

Performance analysis settings cannot be changed during the execution and the results of the analysis are displayed in the **Performance Analysis** window.

REQUIREMENTS FOR PERFORMANCE ANALYSIS

To use performance analysis with your hardware debugger system, you need the appropriate hardware:

- An E1, E2, E2 Lite, EZ-CUBE2, or E20 emulator
- Either a target board with built-in J-Link or a J-Link/J-Link Ultra debug probe and a J-Link RX adapter.

The C-SPY simulator does not support performance analysis.

Analyzing performance

These tasks are covered:

- Using performance analysis.

USING PERFORMANCE ANALYSIS

Getting started analyzing code performance:

- 1 When you have built your application and started C-SPY, choose **C-SPY driver>Performance Analysis** to open the **Performance Analysis** window and click the **Enable** button to turn on the analysis.
- 2 Click the **Setup** button to display the **Performance Analysis Setup** dialog box.
- 3 Use the **Condition** list box to select what to measure and close the dialog box.
- 4 Start executing your application to begin the analysis.
- 5 Measurements are displayed in the **Performance Analysis** window.

You can also choose to set performance breakpoints to measure the execution of certain sections of code. These breakpoints will be displayed in the **Performance Analysis Setup** dialog box.



Before you start a new measurement, you can click the **Clear** button to clear the collected data. To clear just one of the counters, select it before clearing.

Reference information on performance analysis

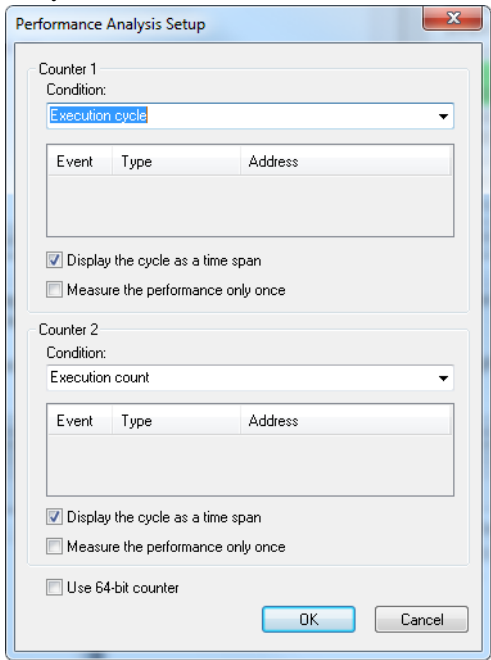
Reference information about:

- *Performance Analysis Setup dialog box*, page 257
- *Performance Analysis window*, page 259
- *Performance Start breakpoints dialog box*, page 261

- *Performance Stop breakpoints dialog box*, page 262.

Performance Analysis Setup dialog box

The **Performance Analysis Setup** dialog box is available from the **Performance Analysis** window and from the **C-SPY driver** menu.



Use this dialog box to configure the analysis. You can configure one or two counters. The size of the counters is 32 bits, which limits the amount of data they can collect. To collect more data, select the **Use 64-bit counter** option and use just one counter.

Requirements

A C-SPY hardware debugger driver.

Condition

Selects what to measure. Choose between:

Not in use

The counter is not in use.

Execution cycle

The number of cycles that have elapsed.

Execution cycle (supervisor mode)

The number of cycles that have elapsed in supervisor mode.

Exception and interrupt cycle

The number of cycles spent processing interrupts and other exceptions.

Exception cycle

The number of cycles spent processing exceptions.

Interrupt cycle

The number of cycles spent processing interrupts.

Execution count

The number of valid instructions executed.

Exception and interrupt count

The number of accepted interrupts and other exceptions.

Exception count

The number of accepted exceptions.

Interrupt count

The number of accepted interrupts.

Display list

This list displays any performance start/stop breakpoints connected to the counter. Information is provided in these columns:

Event

Identifies the breakpoint.

Type

The type of breakpoint: **Start** or **Stop**.

Address

The memory address where the breakpoint is placed.

For information about performance start/stop breakpoints, see *Performance Start breakpoints dialog box*, page 261 and *Performance Stop breakpoints dialog box*, page 262.

Display the cycle as a time span

Converts the number of cycles spent into time and displays the value in the **Time** column of the **Performance Analysis** window, using the operating frequency value from the **Operating Frequency** dialog box; see *Operating Frequency dialog box*, page 62. This option requires that the counter Condition measures cycles.

Measure the performance only once

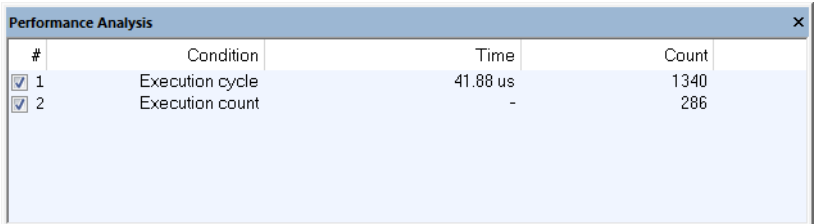
Specifies that the code section between two performance breakpoints is only analyzed once, even if the execution loops. This option can only be used with performance breakpoints.

Use 64-bit counter

Combines the two 32-bit counters to use them as a single 64-bit counter. This increases the capacity when measuring the performance of a single address range. Only the settings from Counter 1 are used.

Performance Analysis window

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



Performance Analysis			
#	Condition	Time	Count
<input checked="" type="checkbox"/> 1	Execution cycle	41.88 us	1340
<input checked="" type="checkbox"/> 2	Execution count	-	286

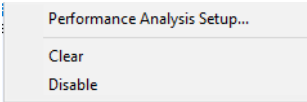
This window displays the performance analysis.

Requirements

A C-SPY hardware debugger driver.

Context menu

This context menu is available:



These commands are available:

Performance Analysis Setup

Displays the **Performance Analysis Setup** dialog box where you configure the measurement. See *Performance Analysis Setup dialog box*, page 257.

Clear

Clears all data in the selected row of the display area. If no row is selected, all data is cleared.

Enable/Disable

Enables or disables the performance analysis. Disabling the analysis does not clear already collected data; re-enabling the analysis and running more passes will append the new data to the previously collected data.

Display area

The display area provides information in these columns:

#

The number of the analysis counter. The numbers are 1 and 2 when a 32-bit counter is used for both of two ranges or 1 when the two 32-bit counters are handled as a 64-bit counter for measurement of performance in a single address range.

The checkboxes enable/disable each counter.

Condition

Indicates the measurement type. See the description for the *Performance Analysis Setup dialog box*, page 257.

Time

The cumulative total of the analyzed execution.

If the measurement type in the **Condition** column is *cycles*, the time is calculated from the operating frequency value and the value in the **Count** column.

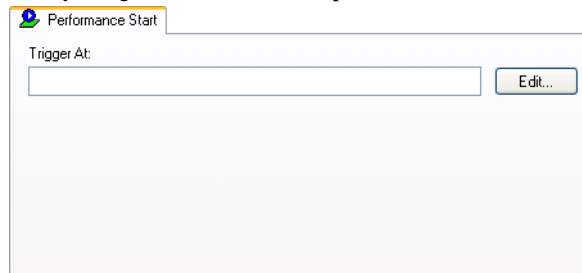
If the measurement type in the **Condition** column is a *count*, this column displays a –.

Count

A decimal value that indicates the number of times the measurement has been performed. Any overflows will be indicated.

Performance Start breakpoints dialog box

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



Use this dialog box to set performance start breakpoints.

To set a Performance Start breakpoint:

- 1** In the editor, **Breakpoints**, or **Disassembly** window, right-click and choose one of the two **Performance Start** commands from the context menu. The number of the breakpoint, 1 or 2, connects the breakpoint to one of the two counters in the **Performance Analysis Setup** dialog box.

Alternatively, to modify an existing breakpoint, select it in the **Breakpoints** window and choose **Edit** on the context menu.
- 2** In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 3** When the breakpoint is triggered, the performance analysis starts.

Requirements

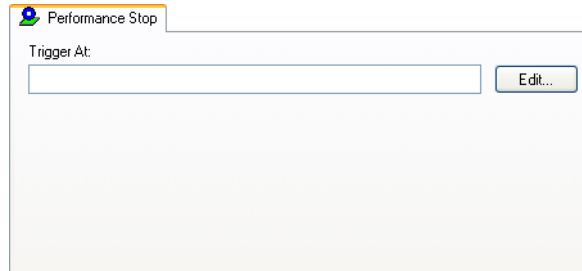
A C-SPY hardware debugger driver.

Trigger At

Specify the location for the breakpoint in the text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 148.

Performance Stop breakpoints dialog box

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



Use this dialog box to set performance stop breakpoints.

To set a Performance Stop breakpoint:

- 1** In the editor, **Breakpoints**, or **Disassembly** window, right-click and choose the **Performance Stop** command that corresponds to a previously defined performance start breakpoint. The number of the breakpoint, 1 or 2, connects the breakpoint to one of the two counters in the **Performance Analysis Setup** dialog box.

Alternatively, to modify an existing breakpoint, select it in the **Breakpoints** window and choose **Edit** on the context menu.
- 2** In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 3** When the breakpoint is triggered, the performance analysis stops.

Requirements

A C-SPY hardware debugger driver.

Trigger At

Specify the location for the breakpoint in the text box. Alternatively, click the **Edit** browse button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 148.

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.

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.

When using code coverage in a hardware debugger system, be aware of the limitations. Code coverage information is based on trace data. Trace data for the E1, E2, E2 Lite, and EZ-CUBE2 emulators and the J-Link debug probe is very limited. If you are using an E1, E2, E2 Lite, or EZ-CUBE2 emulator or a J-Link debug probe, PC

sampling provides better data than any other source. but code coverage will not be complete regardless of the source. Code that has been executed might not be shown as such.

Using code coverage

These tasks are covered:

- Getting started using code coverage


GETTING STARTED USING CODE COVERAGE

To get started using code coverage:

- I 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 13: 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

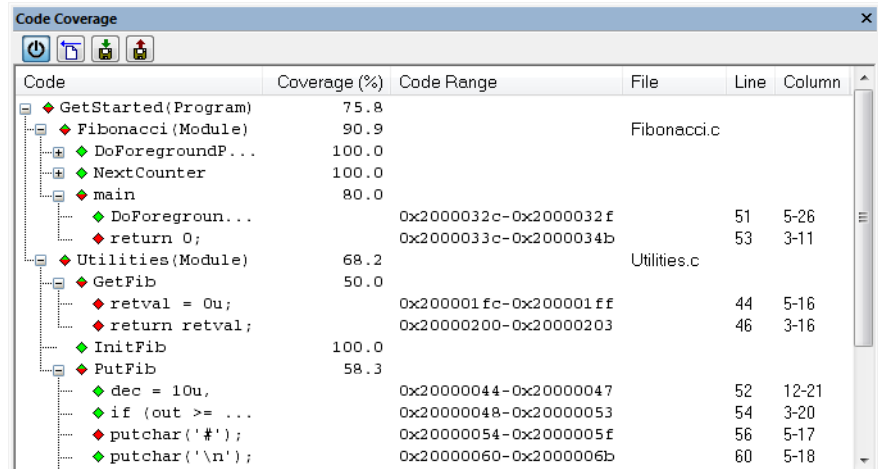
Reference information about:

- *Code Coverage window*, page 265

See also *Single stepping*, page 72.

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

The C-SPY simulator.

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.

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

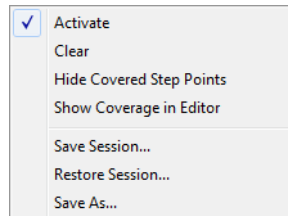
The source file line where the step point is located.

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.

Power debugging

- Introduction to power debugging
- Optimizing your source code for power consumption
- Debugging in the power domain
- Reference information on power debugging

Introduction to power debugging

These topics are covered:

- Reasons for using power debugging
- Briefly about power debugging
- Requirements and restrictions for power debugging

REASONS FOR USING POWER DEBUGGING

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

For examples of when power debugging can be useful, see *Optimizing your source code for power consumption*, page 271.

BRIEFLY ABOUT POWER DEBUGGING

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

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

Measuring power consumption

The debug probe measures the voltage drop across a small resistor in series with the supply power to the device. The voltage drop is measured by a differential amplifier and then sampled by an AD converter.

Power debugging using C-SPY

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

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

REQUIREMENTS AND RESTRICTIONS FOR POWER DEBUGGING

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

- AJ-Link/J-Link Ultra debug probe and a J-Link RX adapter. Target boards with built-in J-Link do not support power debugging.
- An E2 emulator which must be powering the target board. E2 Lite, EZ-CUBE2, and E2 on-board do not support power debugging.

Important! Power measurement for the E2 emulator is based on collecting pairs of current measurements and timestamps after the application execution stops. This slows down debugging performance considerably, so make sure that power logging

is enabled for E2 only when you are actively using the feature. It also means that if the IDE seems to stall now and then, it might be because of this performance reduction.

Optimizing your source code for power consumption

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

These topics are covered:

- Waiting for device status
- Software delays
- DMA versus polled I/O
- Low-power mode diagnostics
- CPU frequency
- Detecting mistakenly unattended peripherals
- Peripheral units in an event-driven system
- Finding conflicting hardware setups
- Analog interference

WAITING FOR DEVICE STATUS

One common construction that could cause unnecessary power consumption is to use a poll loop for waiting for a status change of, for example a peripheral device. Constructions like this example execute without interruption until the status value changes into the expected state.

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

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

SOFTWARE DELAYS

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

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

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

DMA VERSUS POLLED I/O

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

LOW-POWER MODE DIAGNOSTICS

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

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

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

CPU FREQUENCY

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

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

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

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

DETECTING MISTAKENLY UNATTENDED PERIPHERALS

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

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

PERIPHERAL UNITS IN AN EVENT-DRIVEN SYSTEM

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

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



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

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

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

FINDING CONFLICTING HARDWARE SETUPS

To avoid floating inputs, it is a common design practice to connect unused MCU I/O pins to ground. If your source code by mistake configures one of the grounded I/O pins as a logical 1 output, a high current might be drained on that pin. This high unexpected current is easily observed by reading the current value from the Power graph in the

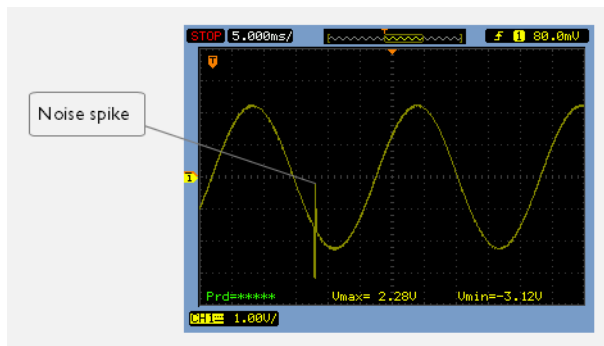
Timeline window. It is also possible to find the corresponding erratic initialization code by looking at the Power graph at application startup.

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

ANALOG INTERFERENCE

When mixing analog and digital circuits on the same board, the board layout and routing can affect the analog noise levels. To ensure accurate sampling of low-level analog signals, it is important to keep noise levels low. Obtaining a well-mixed signal design requires careful hardware considerations. Your software design can also affect the quality of the analog measurements.

Performing a lot of I/O activity at the same time as sampling analog signals causes many digital lines to toggle state at the same time, which might introduce extra noise into the AD converter.



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

Debugging in the power domain

These tasks are covered:

- Displaying a power profile and analyzing the result
- Detecting unexpected power usage during application execution
- Changing the graph resolution

See also:

- *Timeline window—Power graph*, page 285
- *Selecting a time interval for profiling information*, page 247

DISPLAYING A POWER PROFILE AND ANALYZING THE RESULT

To view the power profile:

- 1 Start the debugger.
- 2 Choose **C-SPY driver>Power Log Setup**. Enable power logging and make the required settings. If you are using an E2 emulator, you must also open the **Hardware Setup** dialog box and make sure that the target board is powered by the emulator.
- 3 Choose **C-SPY driver>Timeline** to open the **Timeline** window.
- 4 Right-click in the graph area and choose **Enable** from the context menu to enable the power graph you want to view.
- 5 Choose **C-SPY driver>Power Log** to open the **Power Log** window.
- 6 Optionally, before you start executing your application you can configure the viewing range of the Y-axis for the power graph. See *Viewing Range dialog box*, page 238.
- 7 Click **Go** on the toolbar to start executing your application. In the **Power Log** window, all power values are displayed. In the **Timeline** window, you will see a graphical representation of the power values. For information about how to navigate on the graph, see *Navigating in the graphs*, page 214.
- 8 To analyze power consumption (requires a J-Link debug probe):
 - Double-click on an interesting power value to highlight the corresponding source code in the editor window and in the **Disassembly** window. The corresponding log is highlighted in the **Power Log** window. For examples of when this can be useful, see *Optimizing your source code for power consumption*, page 271.
 - For a specific interrupt, you can see whether the power consumption is changed in an unexpected way after the interrupt exits, for example, if the interrupt enables a power-intensive unit and does not turn it off before exit.
 - For function profiling, see *Selecting a time interval for profiling information*, page 247.

DETECTING UNEXPECTED POWER USAGE DURING APPLICATION EXECUTION

To detect unexpected power consumption:

- 1 Choose **C-SPY driver>Power Log Setup** to open the **Power Log Setup** window.

- 2 In the **Power Log Setup** window, specify a threshold value and the appropriate action, for example **Log All and Halt CPU Above Threshold**.
- 3 Choose **C-SPY driver>Power Log** to open the **Power Log** window. If you continuously want to save the power values to a file, choose **Choose Live Log File** from the context menu. In this case you also need to choose **Enable Live Logging to**.
- 4 Start the execution.

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

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

CHANGING THE GRAPH RESOLUTION

To change the resolution of a Power graph in the Timeline window:

- 1 In the **Timeline** window, select the Power graph, right-click and choose **Open Setup Window** to open the **Power Log Setup** window.
- 2 From the context menu in the **Power Log Setup** window, choose a suitable unit of measurement.
- 3 In the **Timeline** window, select the Power graph, right-click and choose **Viewing Range** from the context menu.
- 4 In the **Viewing Range** dialog box, select **Custom** and specify range values in the **Lowest value** and the **Highest value** text boxes. Click **OK**.
- 5 The graph is automatically updated accordingly.

Reference information on power debugging

Reference information about:

- *Power Log Setup window*, page 278
- *Power Log window*, page 281
- *Timeline window—Power graph*, page 285

See also:

- *Trace window*, page 196
- *The application timeline*, page 211
- *Viewing Range dialog box*, page 238
- *Function Profiler window*, page 249

Power Log Setup window

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

Use this window to configure the power measurement. The contents of the window depends on the debug probe and C-SPY driver you are using.

Note: To enable power logging using a J-Link debug probe, choose **Enable** from the context menu in the **Power Log** window or from the context menu in the power graph in the **Timeline** window.

See also *Debugging in the power domain*, page 275.

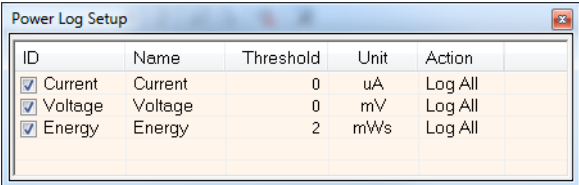
Requirements

One of these alternatives:

- A J-Link/J-Link Ultra debug probe together with a J-Link RX adapter
- An E2 emulator

Display area for the J-Link debug probe

This is what the window looks like for the J-Link/J-Link Ultra debug probe:



ID	Name	Threshold	Unit	Action	
<input checked="" type="checkbox"/> Current	Current	0	uA	Log All	
<input checked="" type="checkbox"/> Voltage	Voltage	0	mV	Log All	
<input checked="" type="checkbox"/> Energy	Energy	2	mWs	Log All	

This area contains these columns:

ID

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

Name

Specify a user-defined name.

Threshold

Specify a threshold value in the selected unit. The action you specify will be executed when the threshold value is reached.

Unit

Displays the selected unit for power. You can choose a unit from the context menu.

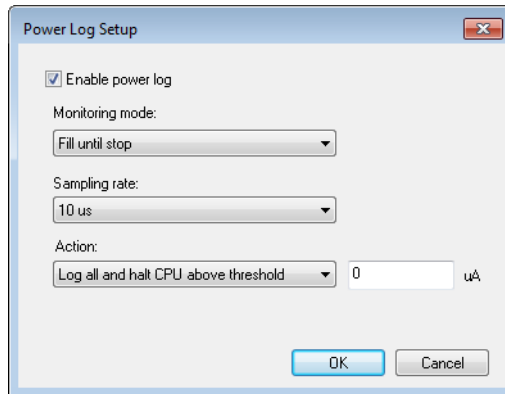
Action

Displays the selected action for the measurement channel. Choose between:

- **Log All**
- **Log Above Threshold**
- **Log Below Threshold**
- **Log All and Halt CPU Above Threshold**
- **Log All and Halt CPU Below Threshold**

Display area for the E2 emulator

This is what the window looks like for the E2 emulator:

**Enable power log**

Enables/disables power logging for the C-SPY E2 emulator driver.

Monitoring mode

Controls the collection of power data. Chose between:

- **Fill until stop**—collects data as long as the application is executing. If the buffer fills up, the oldest data is cleared as new data is written.
- **Fill until full**—collects data until the buffer is full, but continues executing the application.
- **Stop program when full**—stops executing the application when the buffer is full.

Sampling rate

Sets the sampling rate in microseconds.

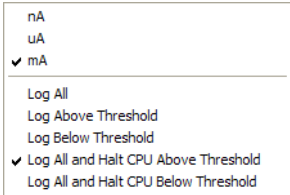
Action

Controls how power data is logged. Choose between:

- **Log all**—logs all collected data.
- **Log all and halt CPU above threshold**—logs all data and stops executing the application when the measured current exceeds the specified value.
- **Log all and halt CPU below threshold**—logs all data and stops executing the application when the measured current falls below the specified value.

Context menu (J-Link only)

This context menu is available for the J-Link/J-Link Ultra debug probe:



These commands are available:

nA, uA, mA

Selects the unit for the power display. These alternatives are available for channels that measure current.

Log All

Logs all values.

Log Above Threshold

Logs all values above the threshold.

Log Below Threshold

Logs all values below the threshold.

Log All and Halt CPU Above Threshold

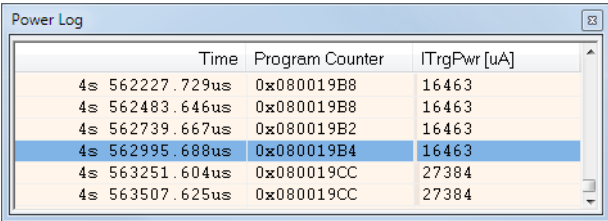
Logs all values. If a logged value exceeds the threshold, execution is stopped. This might take a few execution cycles.

Log All and Halt CPU Below Threshold

Logs all values. If a logged value goes below the threshold, execution is stopped. This might take a few execution cycles.

Power Log window

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



The screenshot shows a window titled "Power Log" with a table containing four columns: Time, Program Counter, and ITrgPwr [uA]. The table has seven rows of data. The first row has a pink header. The second, third, and fourth rows have a pink background. The fifth row is highlighted in blue. The sixth and seventh rows have a pink background.

	Time	Program Counter	ITrgPwr [uA]
4s	562227.729uS	0x080019B8	16463
4s	562483.646uS	0x080019B8	16463
4s	562739.667uS	0x080019B2	16463
4s	562995.688uS	0x080019B4	16463
4s	563251.604uS	0x080019CC	27384
4s	563507.625uS	0x080019CC	27384

This window displays collected power values. This figure shows the **Power Log** window for the J-Link debug probe.

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

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 *Debugging in the power domain*, page 275.

Requirements

One of these alternatives:

- A J-Link/J-Link Ultra debug probe together with a J-Link RX adapter
- An E2 emulator

Display area

This area contains these columns:

Time

The time from the application reset until the event, based on the clock frequency.

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

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

Cycles

The number of cycles from the application reset until the event, based on the operating frequency specified in the **Operating Frequency** dialog box, see *Operating Frequency dialog box*, page 62. This information is cleared at reset.

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

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

Program Counter (J-Link only)

Displays one of these:

An address, which is the content of the PC, that is, the address of an instruction close to where the power value was collected.

---, 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.

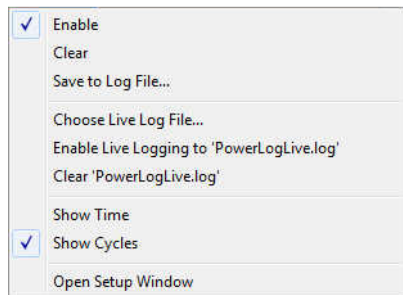
Idle, the power value is logged during idle mode.

Name [unit]

The power measurement value expressed in the unit you specified in the **Power Log Setup** window (J-Link only).

Context menu

This context menu is available:



These commands are available:

Enable

Enables the logging system, which means that power values are saved internally within the IDE. The values are displayed in the **Power Log** window and in the Power graph in the **Timeline** window (if enabled). The system will log information also when the window is closed.

Note: For the E2 emulator, this command only toggles the display of power log data. The power log system can only be enabled/disabled in the **Power Log Setup** window.

Clear

Deletes the log information. Note that this will also happen when you reset the debugger, or if you change the execution frequency in the **Operating Frequency** dialog box.

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.

Choose Live Log File

Displays a standard file selection dialog box where you can choose a destination file for the logged power values. The power values are continuously saved to that file during execution. The content of the live log file is never automatically cleared, the logged values are simply added at the end of the file.

Enable Live Logging to

Toggles live logging on or off. The logs are saved in the specified file.

Clear log file

Clears the content of the live log file.

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.

Open Setup Window

Opens the **Power Log Setup** window.

The format of the log file

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

Time/Cycles

The time from the application reset until the power value was logged.

Approx

An **x** in the column indicates that the power value has an approximative value for time/cycle.

PC

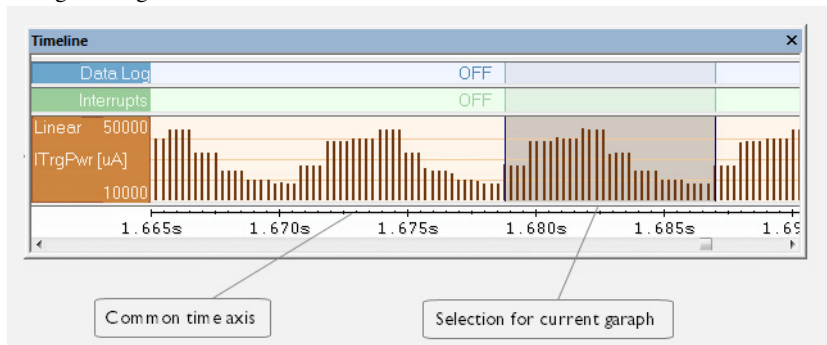
The value of the program counter close to the point where the power value was logged. For the E2 emulator, this will be --- for all values, because the C-SPY E2 emulator driver does not support this feature.

Name [unit]

The corresponding value from the **Power Log** window, where *Name* and *unit* are according to your settings in the **Power Log Setup** window. For the E2 emulator, this will be `Current [mA]` for all values.

Timeline window—Power graph

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



The power graph displays a graphical view of power measurement samples generated by the debug probe or associated hardware in relation to a common time axis.

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 about the **Timeline** window, how to display a graph, and the other supported graphs, see *The application timeline*, page 211.

See also *Requirements and restrictions for power debugging*, page 270.

Requirements

One of these alternatives:

- A J-Link/J-Link Ultra debug probe together with a J-Link RX adapter
- An E2 emulator

Display area

Where:

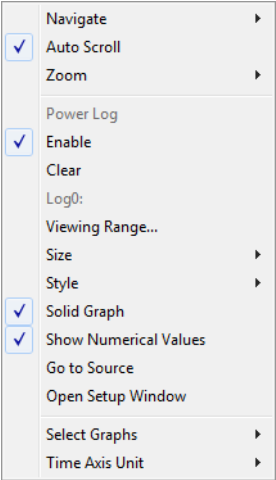
- The label area at the left end of the graph displays the name of the measurement channel.
- The graph itself shows power measurement samples generated by the debug probe or associated hardware.
- The graph can be displayed as a thin line between consecutive logs, as a rectangle for every log (optionally color-filled), or as columns.
- The resolution of the graph can be changed.

- 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.

Power Log

A heading that shows that the Power 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.

Viewing Range

Displays a dialog box, see *Viewing Range dialog box*, page 238.

Size

Determines the vertical size of the graph—choose between **Small**, **Medium**, and **Large**.

Solid Graph

Displays the graph as a color-filled solid graph instead of as a thin line.

Show Numerical Value

Shows the numerical value of the variable, in addition to the graph.

Go To Source

Displays the corresponding source code in an editor window, if applicable.

Open Setup Window

Opens the **Power Log Setup** window.

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. See *Selecting a time interval for profiling information*, page 247.

C-RUN runtime error checking

- Introduction to runtime error checking
- Using C-RUN
- Detecting various runtime errors
- Reference information on runtime error checking
- Compiler and linker reference for C-RUN
- cspybat options for C-RUN

Note that the functionality described in this chapter requires C-RUN, which is an add-on product to IAR Embedded Workbench.

Introduction to runtime error checking

These topics are covered:

- Runtime error checking
- Runtime error checking using C-RUN
- The checked heap provided by the library
- Using C-RUN in the IAR Embedded Workbench IDE
- Using C-RUN in non-interactive mode
- Requirements for runtime error checking

RUNTIME ERROR CHECKING

Runtime error checking is a way of detecting erroneous code constructions when your application is running. This is done by instrumenting parts of the code in the application, or by replacing C/C++ library functionality with a dedicated library that contains support for runtime error checking.

Runtime error checking uses different methods for implementing the checks, depending on the type of your application and in what environment it should run.

Instrumenting the code to perform checks makes the code larger and slower. Variants of library functions with checks will also, in general, be larger and slower than the corresponding functions without checks.

RUNTIME ERROR CHECKING USING C-RUN

C-RUN supports three types of runtime error checking:

- *Arithmetic checking*, which includes checking for integer overflow and underflow, erroneous shifts, division by zero, value-changing conversions, and unhandled cases in switch statements. Normally, the overhead of arithmetic checking is not particularly high, and arithmetic checking can be enabled or disabled on a module by module basis with no complications.
- *Bounds checking*, which checks whether accesses via pointers are within the bounds of the object pointed to. Bounds checking involves instrumenting the code to track pointer bounds, with relatively high costs in both code size and speed. A global table of bounds for indirectly accessed pointers is also needed. You can disable tracking, or just checking, per module or function, but any configuration where pointer bounds are not tracked by all code will usually require some source code adaption.
- *Heap checking using a checked heap*, which checks for errors in the use of heap memory. Heap checking can find incorrect write accesses to heap memory, double free, non-matching allocation and deallocation, and, with explicit calls, leaked heap blocks. Using the checked heap increases the memory size for each heap block, which might mean that you must increase your heap size, and heap operations can take significantly longer than with the normal heap. It also checks only when heap functions are called, which means that it will not catch all heap write errors.

All checks that C-RUN can perform can be used for both C and C++ source code.

You can enable several types of C-RUN checks at the same time. Each type of check that you enable will increase, sometimes very slightly, execution time and code size.

Sometimes, the compiler might merge several checks into one, or move a check out of a loop, in which case the problem may be detected well in advance of the actual access. In these cases, the C-RUN message will display the problem source location (or locations) as separate from the current location.

Before you perform any C-RUN runtime checks, make sure to use all the compiler's facilities for finding problems:

- Do not use Kernighan & Ritchie function declarations—use the prototyped style instead. Read about `--require_prototypes` in the *IAR C/C++ Development Guide for RX*.

- Make sure to pay attention to any compiler warnings before you perform any runtime checking. The compiler will not, in most cases, emit code to check for a problem it has already warned about. For example:

```
unsigned char ch = 1000; /* Warning: integer truncation */
```

Even when integer conversion checking is enabled, the emitted code will not contain any check for this case, and the code will simply assign the value 232 (1000 & 255) to `ch`.

Note: C-RUN depends on the Arm semi-hosting interface (the library function `__iar_ReportCheckFailed` will communicate with C-SPY via the semihosting interface). It is only in non-interactive mode that you can use another low-level I/O interface. See *Using C-RUN in non-interactive mode*, page 292.

For information about how to detect the errors, see *Detecting various runtime errors*, page 295.

THE CHECKED HEAP PROVIDED BY THE LIBRARY

The library provides a replacement *checked heap* that you can use for checking heap usage. The checked heap will insert guard bytes before and after the user part of a heap block, and will also store some extra information (including a sequential allocation number) in each block to help with reporting.

Each heap operation will normally check each involved heap block for changes to the guard bytes, or to the contents of newly allocated heap memory. At certain times (either triggered by a specific call, or after a configurable number of heap operations) a heap integrity check will be performed which checks the entire heap for problems.

It is important to know that the checked heap cannot find erroneous read accesses, like reading from a freed heap block, or reading outside the bounds of an allocated heap block. Bounds checking can find these, as well as many erroneous write accesses that might be missed by the checked heap because they do not write to a guard byte or an otherwise checked byte. The checked heap also checks only when a heap operation is used, and not at the actual point of access.

USING C-RUN IN THE IAR EMBEDDED WORKBENCH IDE

C-RUN is fully integrated in the IAR Embedded Workbench IDE and it offers:

- Detailed error information with call stack information provided for each found error and code correlation and graphical feedback in editor windows on errors
- Error rule management to stop the execution, log, or ignore individual runtime errors, either on project level, file level, or at specific code locations. It is possible to load and save filter setups.

- A bookmark in the editor window for each message which makes it easy to navigate between the messages (using F4).

In the IDE, C-RUN provides these windows:

- The **C-RUN Messages** window, which lists all messages that C-RUN generates. Each message contains a message type (reflecting the check performed), a text that describes the problem, and a call stack. The corresponding source code statements will be highlighted in the editor window. See *C-RUN Messages window*, page 316.
- The **C-RUN Message Rules** window, which lists all rules. See *C-RUN Messages Rules window*, page 318. The rules determine which messages that are displayed in the **C-RUN Messages** window.

USING C-RUN IN NON-INTERACTIVE MODE

You can run C-RUN checked programs using `cspybat—C-SPY` in batch mode. `cspybat` can use rules and other setup configured in the Workbench IDE. C-RUN messages in `cspybat` are by default reported to the host `stdout`, but you can redirect them to a file.

Note: If the module for the report function is inserted into the project, the module should not be compiled with any C-RUN source code options.

The output from `__iar_ReportCheckFailedStdout` is not in user-readable form, as it only contains the raw data. You can use `cspybat` in offline mode (via the options `--rtc_filter` and `--rtc_filter_file`) to transform the raw text into something very similar to normal C-RUN messages.

Use the option `--rtc_enable` to enable C-RUN in `cspybat`. Note that all `cspybat` options for C-RUN all begin with `--rtc_*`. For more information about these options, see *cspybat options for C-RUN*, page 327.

REQUIREMENTS FOR RUNTIME ERROR CHECKING

To perform runtime error checking you need C-RUN, which is an add-on product to IAR Embedded Workbench.

Using C-RUN

These tasks are covered:

- Getting started using C-RUN runtime error checking
- Creating rules for messages

GETTING STARTED USING C-RUN RUNTIME ERROR CHECKING

Typical use of C-RUN involves these steps:

- Determine which C-RUN checks that are needed and specify them in the C-RUN options.
- Run your application in the IAR Embedded Workbench IDE and interactively inspect each C-RUN message. For each message, determine if it is the result of a real problem or not. If not, you can apply a rule to ignore that particular message, or similar messages in the future. If the message is the result of a real problem, you might, depending on the particular circumstances, need to correct the problem and rerun, or you might check for other problems first.
- When finished, close C-SPY. Because the C-RUN windows stay open, now is the time to work through the found problems. Look at the rules setup, possibly edit it, and then save it for future runs.
- Repeat the process until all problems are taken care of.

More in detail, to perform runtime error checking and detect possible runtime errors, follow this example of a typical process:

- 1** To set project options for runtime checking, choose **Project>Options>Runtime Checking** and select the runtime checks you want to perform, for example **Bounds checking**.

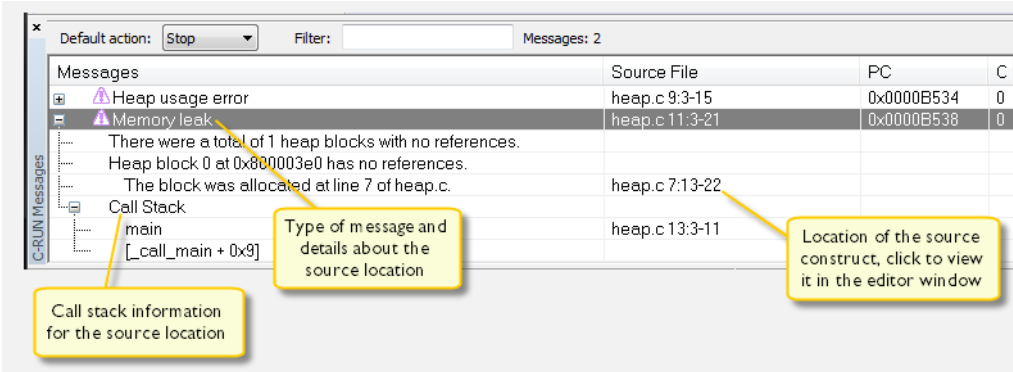
Note that runtime checking must be enabled on the project level, then you must enable each type of check you want to use. Some of the check options, such as **Use checked heap**, and **Enable bounds checking**, must be enabled on the project level, whereas others can be enabled on project or file level.

- 2** Build your application. Note that the lower optimization levels give you better information.
- 3** Start a debug session.
- 4** Start executing your application program.

- 5 If C-RUN detects a possible error, the program execution stops and the corresponding source code is highlighted in the editor window:

```
char *p = malloc(10);  
free(p + 200);  
iar_check_leaks(); // Leakage  
return 0;
```

The C-RUN Messages window is displayed if it is not already open, and it provides information about the source code construct, type of check, and the call stack information for the source location



Note that detection of a problem might not occur at the actual point of access. The check might have been moved out of a loop, or several checks for different accesses might have been merged. In these cases, the problem source (the source for the problem access) might not be in the current statement, and there might be more than one problem source.

- 6 Depending on the source code construct, you might be able to continue program execution after the possible error has been detected. Note that some types of errors might cause unexpected behavior during runtime because of, for example, overwritten data or code.
- 7 If required, use the C-RUN Messages Rules window to specify rules to filter out specific messages based on specific checks and source code locations, specific checks and source files, or specific checks only. You can also specify whether a specific check should not stop the execution, but only log instead. See *Creating rules for messages*, page 295.

You can repeat this procedure for the various runtime checks you want to perform.

CREATING RULES FOR MESSAGES

Depending on your source code, the number of messages in the **C-RUN Messages** window might be very large. For better focus, you can create rules to control which messages you want to be displayed.

To create a rule:

- 1 Select a message in the **C-RUN Messages** window that you want to create a filter rule for.
- 2 Right-click and choose one of the rules from the context menu.
The rule will appear in the **C-RUN Rules** window.
- 3 For an overview of all your rules, choose **View>C-RUN Rules**.

When a check fails, the rules determine how the message should be reported. Rules are scanned top–down and the action from the first matching rule is taken.

Note: You can save a filter setup and then load it later in a new debug session.

Detecting various runtime errors

These tasks are covered:

- Detecting implicit or explicit integer conversion
- Detecting signed or unsigned overflow
- Detecting bit loss or undefined behavior when shifting
- Detecting division by zero
- Detecting unhandled cases in switch statements
- Detecting accesses outside the bounds of arrays and other objects
- Detecting heap usage error
- Detecting heap memory leaks
- Detecting heap integrity violations

Detecting implicit or explicit integer conversion

Description	Checks that an integer conversion (implicit or explicit) or a write access to a bitfield does not change the value.
Why perform the check	Because C allows converting larger types to smaller integer types, some conversions can unintentionally remove significant bits of the value. The check can be limited to implicit

integer conversions, which is useful when the loss of data caused by explicit conversion is considered intentional.

How to use it

Compiler option:

```
--runtime_checking integer_conversion|implicit_integer_conversion
```

In the IDE: **Project>Options>Runtime Checking>Integer conversion**

The check can be applied to one or more modules.

The check can be avoided by inserting an explicit mask:

```
short f(int x)
{
    return x & 0xFFFF; /* Will not report change of value */
}
```

How it works

The compiler inserts code to perform the check at each integer conversion and at each write access to a bitfield, unless the compiler determines that the check cannot fail. Note that an explicit conversion from a constant will not be checked.

Note that increment/decrement operators (++/--) and compound assignments (+=, -=, etc) are checked as if they were written longhand (*var = var op val*).

For example, both ++i and i += 1 are checked as if they were written i = i + 1. In this case, the addition will be checked if overflow checks are enabled, and the assignment will be checked if conversion checks are enabled. For integer types with the same size as int or larger, the conversion check cannot fail. But for smaller integer types, any failure in an expression of this kind will generally be a conversion failure. This example shows this:

```
signed char a = 127;
void f(void)
{
    ++a; /* Conversion check error (128 -> -128) */
    a -= 1; /* Conversion check error (-129 -> 127) */
}
```

The code size increases, which means that if the application has resource constraints this check should be used module per module to minimize the overhead.

Example

Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 293, but use the **Integer conversion** option.

This is an example of source code that will be identified during runtime:

```
int i = 5, j = 0;
char ch = 0;

void conv(void)
{
    ch = i * 100;
}
```

C-RUN will report either Integer conversion failure or Bitfield overflow. This is an example of the message information that will be listed:

Messages		Source File
C-RUN Messages	Integer conversion failure	arith.c 12:8-14
	Conversion changes the value from	500 (0x000001f4)
	to	244 (0xd4).
	Call Stack	
	conv	arith.c 12:3-15
	main	arith.c 27:3-8
	[_call_main + 0x9]	

Detecting signed or unsigned overflow

Description	Checks that the result of an expression is in the range of representable values for its type, and that shift counts are valid. Does not check for overflow in shift operations, which is handled by a separate check. See <i>Detecting bit loss or undefined behavior when shifting</i> , page 299.
Why perform the check	Because the behavior of signed overflow is undefined, and because unsigned overflow results in a truncation that can sometimes be undesirable. Although the shift operation is not checked, shift counts are checked because if a shift count is negative or greater than or equal to the width of the promoted left operand, the behavior of the shift operation is undefined.
How to use it	Compiler option: --runtime_checking signed_overflow unsigned_overflow In the IDE: Project>Options>Runtime Checking>Integer overflow The check can be applied to one or more modules.

The check can be avoided, for example by working in a larger type, when such a type exists:

```
int f(int a, int b)
{ return (int) ((long long) a + (long long) b); }
short g(short a, short b)
{ return (short) ( a + b); } /* Integer promotion occurs */
                             /* assuming that int>16 bits */
```

How it works

The compiler inserts code to perform the check at each integer operation that can overflow (+, -, *, /, %, including unary -) and each shift operation, unless the compiler determines that the check cannot fail.

Note that increment/decrement operators (++/--) and compound assignments (+=, -=, etc) are checked as if they were written longhand (*var = var op val*).

For example, both ++i and i += 1 are checked as if they were written i = i + 1. In this case, the addition will be checked if overflow checks are enabled, and the assignment will be checked if conversion checks are enabled. For integer types with the same size as int or larger, the conversion check cannot fail. But for smaller integer types, any failure in an expression of this kind will generally be a conversion failure. This example shows this:

```
signed char a = 127;
void f(void)
{
    ++a;    /* Conversion check error (128 -> -128) */
    a -= 1; /* Conversion check error (-129 -> 127) */
}
```

The code size increases, which means that if the application has resource constraints this check should be used per module to minimize overhead.


Example

Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 293, but use the **Integer overflow** option.

This is an example of source code that will be identified during runtime:

```
unsigned long ovfl(void)
{
    unsigned long ul = i + 0x7fffffff;
    return ul;
}
```

C-RUN will report either Signed integer overflow, Unsigned integer overflow, or Shift count overflow. This is an example of the message information that will be listed:

Messages		Source File
C-RUN Messages	 Signed integer overflow	arith.c 26:22-35
	Result is greater than the largest representable number: 5 (0x5) + 2147483647 (0x7fffffff).	
	Call Stack	
	ovfl	arith.c 26:17-36
	main	arith.c 35:3-8
	[_call_main + 0x9]	


Detecting bit loss or undefined behavior when shifting

Description	Checks for overflow in shift operations and that shift counts are valid.
Why perform the check	<p>Because the behavior of signed overflow is undefined, and because unsigned overflow results in a truncation that can sometimes be undesirable.</p> <p>Overflow occurs in a left shift operation $E1 \ll E2$ if $E1$ is negative or if the result, defined as $E1 * 2^{E2}$, is not in the range of representable values for its type.</p>
How to use it	<p>Compiler option: <code>--runtime_checking signed_shift unsigned_shift</code></p> <p>In the IDE: Project>Options>Runtime Checking>Integer shift overflow</p> <p>The check can be applied to one or more modules.</p> <p>The check can be avoided by masking before shift:</p> <pre>/* Cannot overflow */ int f(int x) { return (x & 0x00007FFF) << 16; }</pre>
How it works	<p>The compiler inserts code to perform the check for each shift operation, unless the compiler determines that the check cannot fail.</p> <p>The code size increases, which means that if the application has resource constraints this check should be used per module to minimize the overhead.</p>
Example	<p>Follow the procedure described in <i>Getting started using C-RUN runtime error checking</i>, page 293, but use the Integer shift overflow option.</p>

This is an example of source code that will be identified during runtime:

```
void shift(void)
{
    i <<= 31;
}
```

C-RUN will report either `Shift overflow` or `Shift count overflow`. This is an example of the message information that will be listed:

Messages		Source File
C-RUN Messages	 Shift overflow	arith.c 32:3-10
	Result is greater than the largest representable number:	
	signed value 5 (0x5) doubled 31 time(s).	
	Cell Stack	
	shift	arith.c 32:3-11
	main	arith.c 41:3-9
	[_call_main + 0x9]	


Detecting division by zero

Description	Checks for division by zero and modulo by zero. Floating-point operations are checked for division by exactly (positive) zero.
Why perform the check	Because the behavior of integer division by zero is undefined, and because floating-point division by exactly zero usually indicates a problem.
How to use it	Compiler option: <code>--runtime_checking div_by_zero</code> In the IDE: Project>Options>Runtime Checking>Division by zero The check can be applied to one or more modules.
How it works	The compiler inserts code to perform the check at each division and modulo operation, unless the compiler determines that the check cannot fail.
Example	Follow the procedure described in <i>Getting started using C-RUN runtime error checking</i> , page 293, but use the Division by zero option.

This is an example of source code that will be identified during runtime:

```
void div(void)
{
    j = i / j;
}
```

C-RUN will report `Division by zero`. This is an example of the message information that will be listed:

Messages	Source File
 Division by zero	arith.c 7:7-11
Division by zero.	
Call Stack	
div	arith.c 7:3-12
main	arith.c 37:3-7
[_call__main + 0x9]	

Detecting unhandled cases in switch statements

Description	Checks for a missing <code>case</code> label in a <code>switch</code> statement that does not have a default label.
Why perform the check	The check is useful, for example, to detect when an <code>enum</code> type has been augmented with a new value that is not yet handled in a <code>switch</code> statement.

- How to use it

Compiler option: `--runtime_checking switch`

In the IDE: **Project>Options>Runtime Checking>Switch**

The check can be applied to one or more modules.

The check can be avoided by adding a default label.
- How it works

The compiler inserts an implicit default label to perform the check in each `switch` statement that does not have a default label.
- Example

Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 293, but use the **Switch** option.

This is an example of source code that will be identified during runtime:

```
void sw(void)
{
    switch(ch)
    {
        case 0: i = 3; break;
        case 5: i = 2; break;
    }
}
```

C-RUN will report `Unhandled case in switch`. This is an example of the message information that will be listed:

Messages	Source File
Unhandled case in switch	arith.c 17:3-12
Switch to undefined case label.	
Cell Stack	
sw	arith.c 22:1-1
main	arith.c 39:3-6
[_call_main + 0x9]	

Detecting accesses outside the bounds of arrays and other objects

- Description

Checks that accesses through pointer expressions are within the bounds of the expected object. The object can be of any type and can reside anywhere—globally, on the stack, or on the heap.

Why perform the check The check is useful whenever your application reads or writes to locations it should not. For example:

```
int arr[10] = {0};
int f(int i)
{
    return arr[i];
}
int g(void)
{
    return f(20); /* arr[20 is out of bounds] */
}
```

How to use it Compiler option: `--runtime_checking bounds`

In the IDE: **Project>Options>Runtime Checking>Enable bounds checking**

This will enable out-of-bounds checking globally. Note that there are suboptions that you can use to fine-tune the out-of-bounds checking globally and for each source file.

How it works In code where pointer bounds are tracked:

- Each transfer of a pointer value also transfers the bounds for that pointer value.
- When a pointer is initialized to point to an object of some sort, the bounds of the pointer are set to the bounds of the object. If the object is an array, the bounds cover the entire array. If it is a single instance, the bounds cover the single instance.
- When a pointer is initialized to an absolute address, the pointer is assumed to point to a single object of the specified type. For example:

```
uint32_t * p = (uint_32_t *)0x100;
```

In this case, `p` will point to a 32-bit unsigned integer at address `0x100`, with the bounds `0x100` and `0x104`.

- A null pointer is given bounds that do not cover any access, in other words, an access through it is erroneous.
- When a pointer value is passed to a function as a parameter, the bounds are passed as extra, hidden, parameters.
- When a pointer value is returned from a function, the returned value and the bounds are passed in a `struct` as the actual return value.
- When a pointer value is stored in memory in such a way that it can be accessed via pointers, its bounds are stored in a global bounds table. Whenever the pointer value is accessed, the associated bounds in the global bounds table are retrieved as well. The size of the global bounds table can be changed using **Number of entries** (the linker option `--bounds_table_size number_of_records[:number_of_buckets] | (number_of_bytes)`).

- In other cases, the bounds are kept track of in extra local variables.

For each access through a pointer expression, the calculated address and the calculated address plus the access size is checked against the bounds. If any of the two addresses are outside of the bounds, a C-RUN message is generated.

Functions that receive pointers in any parameters, or that return a pointer value, can exist in two variants, one with the bounds, and one without the bounds.

Resource usage

The bounds checking overhead can cause the application to no longer fit in the available ROM or RAM. There are some ways you can try to deal with this:

- Provided that your application does not use too many indirectly accessed pointers, you can shrink the global bounds table to reduce the amount of RAM used for it. See `--bounds_table_size`, page 321 (in the IDE, **Number of entries**).
By default, 4-Kbyte entries that need about 48 Kbytes are used.
- You can turn off the actual bounds checks in some modules. This will reduce the amount of code added by instrumentation to some extent.
- You can turn off pointer bounds tracking in some modules. This will eliminate the increase in code size entirely in these modules, but will cause problems in the interface between the code that does track pointer bounds and the code that doesn't. See the next section for more about this.

Non-checked code

Sometimes you cannot enable bounds checking in the entire application, for example if some part of the application is an externally built library, or is written in assembler or inline assembler. If you add any extra source code lines to make your code work for bounds checking, use the preprocessor symbol `__AS_BOUNDS__` to make the extra source code conditional. These are some cases you should consider:

- Calling code that does not track bounds from code that does

This only affects functions with pointers as parameters or as return types.

By using `#pragma no_bounds` or `#pragma default_no_bounds` on your declarations, you can specify that certain functions do not track pointer bounds. If you call such a function from code that does not track pointer bounds, no extra hidden parameters are passed, and any returned pointers are either considered “unsafe” (all checked accesses via such pointers generate errors) or “safe” (accesses via such pointers cannot fail), depending on whether the option **Check pointers from non-instrumented functions** has been used or not (compiler option `--ignore_uninstrumented_pointers`). If you wish to explicitly specify the bounds on such values, use the built in operator `__as_make_bounds`.

For example:

```
#pragma no_bounds
struct X * f1(void);
...
{
    struct X *px = f1();
    /* Set bounds to allow accesses to a single X struct.
       (If the pointer can be NULL, you must check for that.) */
    if (px)
        px = __as_make_bounds(px, 1);
    /* From here, any accesses via the pointer will be checked
       to ensure taht they are within the struct. */
}
```

- Calling code that tracks bounds from code that does not

If you call a function that tracks bounds, and which has pointers as parameters, or which returns a pointer, from code that does not track bounds, you will generally get an undefined external error when linking. To enable such calls, you can use

`#pragma generate_entry_without_bounds` or the option **Generate functions callable from non-instrumented code** (compiler option

`--generate_entries_without_bounds`) to direct the compiler to emit one or more extra functions that can be called from code that does not track bounds. Each such function will simply call the function with default bounds, which will be either "safe" (accesses via such pointers never generate errors) or "unsafe" (accesses via such pointers always generate errors) depending on whether the option **Check**

pointers from non-instrumented functions (compiler option

`--ignore_uninstrumented_pointers`) has been used or not.

If you want to specify more precise bounds in this case, use

```
#pragma define_without_bounds.
```

You can use this pragma directive in two ways. If the function in question is only called from code that does not track pointer bounds, and the bounds are known or can be inferred from other parameters, there is no need for two functions, and you can simply modify the definition using `#pragma define_without_bounds`.

For example:

```
#pragma define_without_bounds
int f2(int * p, int n)
{
    p = __as_make_bounds(p, n); /* Give p bounds */
    ...
}
```

In the example, `p` is assumed to point to an array of `n` integers. After the assignment, the bounds for `p` will be `p` and `p + n`.

If the function can be called from both code that does track pointer bounds and from code that does not, you can instead use `#pragma define_without_bounds` to

define an extra variant of the function without bounds information that calls the variant with bounds information.

You cannot define both the variant without bounds and the variant with bounds in the same translation unit.

For example:

```
#pragma define_without_bounds
int f3(int * p, int n)
{
    return f3(__as_make_bounds(p, n), n);
}
```

In the example, `p` is assumed to point to an array of `n` integers. The variant of `f3` without extra bounds information defined here calls the variant of `f3` with extra bounds information ("`f3 [with bounds]`"), giving the pointer parameter bounds of `p` and `p + n`.

- Global variables with pointers defined in code that does not track bounds

These pointers will get either bounds that signal an error on any access, or, if the option **Check pointers from non-instrumented memory** (linker option `--ignore_uninstrumented_pointers`) is used when linking, bounds that never cause an error to be signaled. If you need more specific bounds, use `__as_make_bounds`.

For example:

```
extern struct x * gp_ptr;
int main(void)
{
    /* Give gp_ptr bounds with size N. */
    gp_ptr = __as_make_bounds(gp_ptr, N);
    ...
}
```

- RTOS tasks

The function that implements a task might get called with a parameter that is a pointer. If the RTOS itself is not tracking pointer bounds, you must use `#pragma define_without_bounds` and `__as_make_bounds` to get the correct bounds information.

For example:

```
#pragma define_without_bounds
void task1(struct Arg * p)
{
    /* p points to a single Arg struct */
    p = __as_make_bounds(p, 1);
    ...
}
```

Some limitations

- Function pointers

Sharing a function pointer between code that tracks bounds and code that does not can be problematic.

There is no difference in type between functions that track bounds, and functions that do not. Functions of both kinds can be assigned to function pointers, or passed to functions that take function pointer parameters. However, if a function whose signature includes pointers is called in a non-matching context (a function that tracks bounds from code that does not, or vice versa), things will not work reliably. In the most favorable cases, this will mean confusing bounds violations, but it can cause practically any behavior because these functions are being called with an incorrect number of arguments.

For things to work, you must ensure that all functions whose signature includes pointers, and which are called via function pointers, are of the right kind. For the simple case of call-backs from a library that does not track bounds, it will usually suffice to use `#pragma no_bounds` on the relevant functions.

- K&R functions

Do not use K&R functions. Use `--require_prototypes` and shared header files to make sure that all functions have proper prototypes. Note that in C `void f()` is a K&R function, while `f(void)` is not.

- Pointers updated by code that does not track bounds

Whenever a pointer is updated by code that does not set up new bounds for the pointer, there is a potential problem. If the new pointer value does not point into the same object as the old pointer value, the bounds will be incorrect and an access via this pointer in checked code will signal an error.

- `low_level_init`

C-RUN cannot manage pointers in `low_level_init`. This means that you must compile the module—and any module it refers to—that contains `low_level_init` without C-RUN enabled.

- Address zero

Dereferencing a pointer variable that contains the address zero is a C-RUN error.

- Bounds-checking does not work with atomic access functions that either take pointers as parameters or return a pointer. To make bounds-checking work in these cases, encapsulate the atomic access function in a no-bounds-checking function.
- Because the bounds-checking table is not thread-safe, C-RUN bounds-checking will generally not work with threaded applications if bounds-checking is turned on in more than one thread.

Absolute addresses

If you use `#pragma location` or the `@` operator to place variables at absolute addresses, pointers to these variables will get correct bounds, just like pointers to any other variables.

If you use an explicit cast from an integer to a pointer, the pointer will get bounds assuming that it points to a single object of the specified type. If you need other bounds, use `__as_make_bounds`.

For example:

```
/* p will get bounds that assume it points to a single struct
   Port at address 0x1000. */
p = (struct Port *)0x1000;
/* If it points to an array of 3 struct you can add */
p = __as_make_bounds(p, 3);
```

Example

Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 293, but use the **Bounds checking** option.

This is an example of source code that will be identified during runtime:

```
int Arr[4] = { 0, 1 ,2, 3};

int ArrI = 5;

int f(void)
{
    int i = Arr[ArrI + 1]; // Double fail global
    i += Arr[ArrI + 2];
    return i;
}
```

C-RUN will report either **Access out of bounds** or **Invalid function pointer**.

This is an example of the message information that will be listed:

Messages		Source File
C-RUN Messages	Access out of bounds	file.c 9:11-23, fil
	Access outside pointer bounds:	
	Access 0x80000038 - 0x80000040	
	Bounds 0x80000020 - 0x80000030, int Arr[4];	file.c 3:5-7
	Call Stack	
	f	file.c 9:7-24
	main	file.c 29:8-10
	[_call_main + 0x9]	

Note: The ranges displayed for accesses and bounds include the start address but not the end address.

Detecting heap usage error

Description	<p>Checks that the heap interface—<code>malloc</code>, <code>new</code>, <code>free</code>, etc—is used properly by your application. The following improper uses are checked for:</p> <ul style="list-style-type: none"> Using the incorrect deallocator—<code>free</code>, <code>delete</code>, etc—for an allocator—<code>malloc</code>, <code>new</code>, etc. For example: <pre>char * p1 = (char *)malloc(23); /* Allocation using malloc. */ char * p2 = new char[23]; /* Allocation using new[]. */ char * p3 = new int; /* Allocation using new. */ delete p1 /* Error, allocated using malloc. */ free(p2); /* Error, allocated using new[]. */ delete[] p3; /* Error, allocated using new. */</pre> Freeing a heap block more than once. Trying to allocate a heap block that is too large.
Why perform the check	To verify that the heap interface is used correctly.
How to use it	<p>Linker option: <code>--debug_heap</code></p> <p>In the IDE: Project>Options>Runtime Checking>Use checked heap</p> <p>The checked heap will replace the normal heap for the whole application. The checked heap requires extra heap and stack resources. Make sure that your application has at least 10 Kbytes of heap and 4 Kbytes of stack.</p> <p>The limit for how large a heap block can be at allocation is by default 1 Mbyte. The limit can be changed by the function:</p> <pre>size_t __iar_set_request_report_limit(size_t value);</pre> <p>The function returns the old limit. You can find the declaration of this function in <code>iar_dlmalloc.h</code>. For more information, see the <i>IAR C/C++ Development Guide for RX</i>.</p>
How it works	<p>For any incorrect use of the heap interface, a message will be issued.</p> <p>See also <i>The checked heap provided by the library</i>, page 291.</p>
Example	Follow the procedure described in <i>Getting started using C-RUN runtime error checking</i> , page 293, but use the Debug heap option.

This is an example of source code that will be identified during runtime:

```
int main(void)
{
    char *p = malloc(10);
    free(p + 200);
    iar_check_leaks(); // Leakage
    return 0;
}
```

C-RUN will report either Heap integrity violation or Heap usage error. This is an example of the message information that will be listed:

Messages		Source File
C-RUN Messages	Heap usage error	heap.c 9:3-15
	The address 0x800004a8 does not appear to be the start of a h...	
	Call Stack	
	main	heap.c 11:3-21
	[_call_main + 0x9]	

Detecting heap memory leaks

Description	Checks for heap blocks without references at a selected point in your application.
Why perform the check	<p>A leaked heap block cannot be used or freed, because it can no longer be referred to. Use this check to detect references to heap blocks and report blocks that are seemingly unreferenced. Note that the leak detection cannot find all possible memory leak cases, a seemingly unreferenced heap block might actually be referenced and a seemingly referenced heap block might actually be leaked.</p> <p>Note: The leak checker does not currently support multi-threaded environments. The leak checker works by scanning known RAM locations for references to heap blocks. The thread executing the leak check has information about its own stack, but not about the stack of other threads. The missing information can result in both false positives and false negatives.</p>
How to use it	<p>Linker option: <code>--debug_heap</code></p> <p>In the IDE: Project>Options>Runtime Checking>Use checked heap</p> <p>The checked heap will replace the normal heap for the whole application. The checked heap requires extra heap and stack resources. Make sure that your application has at least 10 Kbytes of heap and 4 Kbytes of stack.</p>

The leak detection check must be called manually. It can either be called at the exit of the application or it can be used for detecting leaked heap blocks between two source points. These functions are defined in `iar_dlmalloc.h`:

- `void __iar_leaks_ignore_all(void);`
Use this function to mark all currently allocated heap blocks to be ignored in subsequent heap leakage checks.
- `void __iar_leaks_ignore_block(void *block);`
Use this function to mark a specific allocated heap block to be ignored in subsequent heap leakage checks.
- `void __iar_check_leaks(void);`
Use this function to check for leaks.

How it works

The checked heap will replace the normal heap for the whole application. The heap leakage algorithm has three phases:

- 1 Scans the heap and makes a list of all allocated heap blocks.
- 2 Scans the statically used RAM, the stack, etc for addresses in the heap. If the address matches one of the heap blocks in the list above, it is removed from the list.
- 3 Reports the remaining heap blocks in the list as leaked.

See *The checked heap provided by the library*, page 291.

Example

Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 293, but use the **Debug heap** option.


This is an example of source code that will be identified during runtime:

```
char *p = malloc(10);
p = malloc(20);

__iar_check_leaks();

return *p;
```

C-RUN will report `Memory leak`. This is an example of the message information that will be listed:

Messages	Source File
 Memory leak	heap_leak1.c 188:3-21
There were a total of 1 heap blocks with no references.	
Heap block 0 at 0x2000c670 has no references.	
The block was allocated at line 185 of heap_leak1.c.	heap_leak1.c 185:13-22
Cell Stack	
main	heap_leak1.c 190:3-12
[_cell_main + 0x9]	

Detecting heap integrity violations

Description	<p>Checks for various heap integrity violations. The check can either be manually triggered or can be set up to be triggered at regular intervals of use of the heap interface. Integrity problems that can be detected when you enable this check are:</p> <ul style="list-style-type: none"> ● Destruction of the internal heap structure. Mostly, this is because a write access through a pointer expression is incorrect. Use out-of-bounds checking to try to locate the erroneous write access. ● Write accesses outside allocated memory, for example: <pre>char * p = (char *)malloc(100); /* Memory is allocated. */ ... p[100] = ... /* This write access is out of bounds. */</pre> <p>A write access that is out-of-bounds of the heap block and that changes the guards in front of or after the heap block will be detected. Any other write accesses will not be detected.</p> ● Write accesses to freed memory, for example: <pre>char * p = (char *)malloc(...); /* Memory is allocated. */ ... free(p); /* Memory is freed. */ ... p[...] = ... /* Write access to freed memory. */</pre> <p>If the memory that contains the original <code>p</code> is allocated again before <code>p</code> is written to, this error will typically not be detected. By using the delayed free list (see below), this error can be found.</p>
Why perform the check	<p>Use the checked heap if you suspect that your application, at some point, writes erroneously in the heap, for example by misusing a heap block.</p>
How to use it	<p>Linker option: <code>--debug_heap</code></p> <p>In the IDE: Project>Options>Runtime Checking>Use checked heap</p> <p>The checked heap will replace the normal heap for the whole application. The checked heap requires extra heap and stack resources. Make sure that your application has at least 10 Kbytes of heap and 4 Kbytes of stack.</p> <p>For detecting heap integrity violations, you can use these functions which are defined in <code>iar_dlmalloc.h</code>:</p> <ul style="list-style-type: none"> ● <code>size_t __iar_check_heap_integrity(void);</code> Use this function to verify the integrity of the heap. If any corruptions are detected, they are reported. The return value is the number of found problems. There is a limit on the number of corruption errors that are reported. This limit can be changed by

using the `__iar_set_integrity_report_limit` function. Execution is only stopped when the final message is generated. The default number of reported messages is 10. A call to `__iar_check_heap_integrity` is not guaranteed to return to the caller if the heap is corrupt.

- `size_t __iar_set_heap_check_frequency(size_t interval);`

Use this function to specify how often the periodic heap integrity checks are performed. By default, the periodic checks are turned off (`interval = 0`). If `interval` is a positive number, the integrity will be checked every `interval`:th heap operation where every call to `free/malloc/new/delete/realloc/etc` counts as one operation. The function returns the old interval, which means that the state can be restored if necessary. The heap check interval can be increased or turned off when trusted parts of your application program, and then be decreased when you run parts of your application that are likely to contain heap errors.

- `size_t __iar_set_delayed_free_size(size_t size);`

Use this function to specify the maximum size of the freed delay list. By default, the freed delay list is turned off (`size = 0`). This function has no effect on the actual size of the list, it only changes the maximum. The function returns the previous value so it can be restored if necessary.

The freed delay list can be used to try to find locations in your application that use a freed heap block. This can help you detect:

- Mixing up an old heap block pointer that has been freed with a new, freshly allocated heap block pointer. Because the freed delay list will delay the actual reuse of a freed heap block, the behavior of your application might change and you might be able to detect the presence of this kind of problem.
- Writes to already freed heap blocks. If a heap block is in the freed delay list, it will get specific content, different from when it is actually freed, and a heap integrity check can find those erroneous write accesses to the heap block.
- `size_t __iar_free_delayed_free_size(size_t count);`

Use this function to make sure that at most `count` elements are present in the freed delay list. Superfluous elements are freed (the oldest ones change first). It has no effect on the maximum size of the list—it only changes the current number of elements. Calling this function has no effect if `count` is larger than the current size of the list. The function returns the number of freed elements.

How it works

The checked heap will replace the normal heap for the whole application.

The *freed delay list* is a queuing mechanism for `free` calls. When calling `free`, or an equivalent memory operation that returns memory to the heap, the recently freed pointer is *queued* to be freed instead of actually *being* freed. If the maximum size of the delay list is exceeded, the oldest elements above the maximum size in the freed delay list are actually freed.

All errors that the checked heap reports, mention a heap block that is somehow corrupt. The checked heap cannot inform about who corrupted the heap block or when it was corrupted. You can use calls to the `__iar_debug_check_heap_integrity` function to verify the integrity during application execution and narrow down the list of potential candidates.

For example:

```
...
__iar_debug_check_heap_integrity(); /* Pre-check */
my_function(..., ..., ...);
__iar_debug_check_heap_integrity(); /* Post-check */
...
```

If the post-check reports problems that the pre-check does not, it is probable that `my_function` corrupted the heap.

The checked heap consumes resources:

- The checked heap requires more ROM space than the normal heap implementation
- All heap operations require more time in the checked heap
- Each heap block in the checked heap contains additional space for bookkeeping, which results in increased RAM usage for your application.

See *The checked heap provided by the library*, page 291.

Example


Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 293, but use the **Checked heap** option.

This is an example of source code that will be identified during runtime:

```
void check(void)
{
    char *p = malloc(10);

    p[11] = 1;
    __iar_check_heap_integrity();
}
```

C-RUN will report Heap integrity violation. This is an example of the message information that will be listed:

Messages	Source File
 Heap integrity violation 1 heap integrity errors were detected. Violation detected in heap block 1 at address 0x80000408. The block was allocated at line 7 of heap.c.	heap.c 10:3-30
Call Stack check main [_call_main + 0x9]	heap.c 7:13-22 heap.c 11:1-1 heap.c 21:3-9

Reference information on runtime error checking

Reference information about:

- *C-RUN Runtime Checking options*, page 314
- *C-RUN Messages window*, page 316
- *C-RUN Messages Rules window*, page 318

C-RUN Runtime Checking options

The **C-RUN Runtime Checking** options determine which checks to perform at runtime.

C-RUN Runtime Checking

☒ **Enable**

☐ Use checked heap☐ Enable bounds checking

Instrumentation☒ Track pointer bounds☒ Check accesses☐ Generate functions callable from non-instrumented code☒ Check pointers from non-instrumented functions

Global bounds table☒ Check pointers from non-instrumented memory

Number of entries: 1000

Insert checks for☐ Integer overflow☐ Including unsigned☐ Integer conversion☐ Including explicit casts☐ Integer shift overflow☐ Including unsigned shifts☐ Division by zero☐ Unhandled switch case

See also *Using C-RUN*, page 292.

Enable

Enables runtime checking.

Use checked heap

Uses the checked heap, to detect heap usage errors.

Enable bounds checking

Checks for accesses outside the bounds of arrays and other objects. Available checks:

Track pointer bounds

Makes the compiler add code that tracks pointer bounds. If you want to check pointer bounds, you should enable **Check accesses** and then decide how instrumented code should interact with non-instrumented code:

Check accesses

Inserts code for checking accesses via pointers.

Generate functions callable from non-instrumented code	When Track pointer bounds is enabled, any functions that return or receive types that contain pointers are modified to also return/receive pointer bounds. Use this option to generate an extra entry for each such function, which can be called from unchecked code.
Check pointers from non-instrumented functions	<p>When Track pointer bounds is enabled, pointers that originate from functions that are not instrumented for bounds checking are by default given globally permissive bounds information. Use this option to identify these pointers—any accesses via such pointers will generate an error. In this way you can manually replace the globally permissive bounds information with valid counterparts, see <code>__as_get_base</code>, page 326, <code>__as_get_bound</code>, page 326, <code>__as_make_bounds</code>, page 326.</p> <p>If this option is not used and you do not specify valid bounds information, accesses via such pointers do not generate errors and might result in unnoticed incorrect runtime behavior.</p>
Check pointers from non-instrumented memory	<p>When Track pointer bounds is enabled, each time a pointer is loaded from memory, its bounds are looked up in the global bounds table. If no entry is found in the table for this pointer, usually because the pointer was created by non-instrumented code, it is given globally permissive bounds. Use this option to identify such pointers—any accesses via such pointers will generate an error. In this way you can manually replace the globally permissive bounds information with valid counterparts, see <code>__as_get_base</code>, page 326, <code>__as_get_bound</code>, page 326, <code>__as_make_bounds</code>, page 326.</p> <p>If this option is not used and you do not specify valid bounds information, accesses via such pointers do not generate errors and might result in unnoticed incorrect runtime behavior.</p>

Number of entries

The bounds checking system uses a separate table to track bounds for pointers in memory. Use this option to set the number of such bounds that can be tracked simultaneously. The table will use approximately 50 bytes per pointer.

Insert checks for

Inserts checks for:

Integer overflow

Checks for signed overflow in integer operations. Use **Including unsigned** to also check for unsigned overflow in integer operations.

Integer conversion

Checks for implicit integer conversions resulting in a change of value. Use **Including explicit casts** to also check for explicit casts.

Integer shift overflow

Checks for overflow in shift operations. Use **Including unsigned shifts** to also check for unsigned overflow in shift operations.

Division by zero

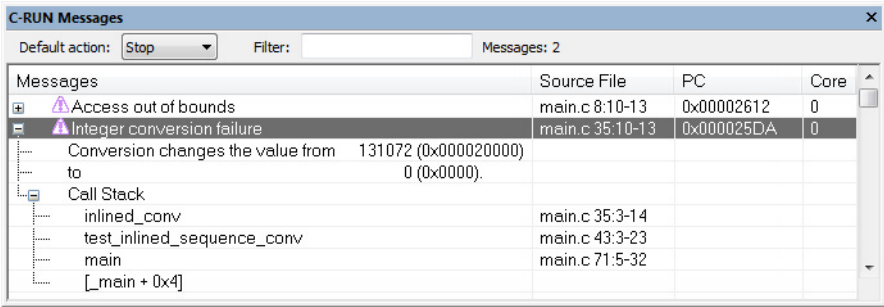
Checks for division by zero.

Unhandled switch case

Checks for unhandled cases in `switch` statements

C-RUN Messages window

The C-RUN Messages window is available from the **View** menu.



This window displays information about runtime errors detected by a runtime check. The window groups messages that have the same source statement, the same call stack, and the same messages.

See also *Using C-RUN*, page 292.

Requirements

A license for the C-RUN product.

Toolbar

The toolbar contains:

Default action

Sets the default action for what happens if no other rule is satisfied. Choose between **Stop**, **Log**, and **Ignore**.

Filter

Filters the list of messages so that only messages that contain the text you specify will be listed. This is useful if you want to search the message text, call stack entries, or filenames.

Messages

Lists the number of C-RUN messages.

Display area

The display area shows all detected errors since the last reset. More specifically, the display area provides information in these columns:

Messages

Information about the detected runtime error. Each message consists of a headline, detailed information about the error, and call stack information for the error location. Note that ranges displayed for accesses and bounds include the start address but not the end address.

Source File

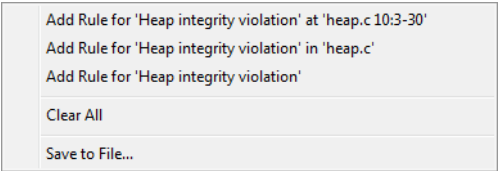
The name of the source file in which a runtime error was detected, or otherwise a relevant location, for example variable definitions.

PC

The value of PC when the runtime error was detected.

Context menu

This context menu is available:



These commands are available:

Add Rule for ... at *range*

Adds a rule that matches this particular runtime check at this particular location.

Add Rule for ... in *filename*

Adds a rule that matches all runtime checks of this kind in the specified file.

Add Rule for ...

Adds a rule that matches all runtime checks of this kind.

Clear All

Clears the window from all content.

Save to File

Opens a dialog box where you can choose to save content to a file, either in text or XML format.

C-RUN Messages Rules window

The C-RUN Messages Rules window is available from the **View** menu.

C-RUN Message Rules	Check	Source File	Action	
	Memory leak	heap.c 19:3-21	Ignore	v
	Heap usage error	heap.c 17:3-15	Ignore	v
	Heap usage error	heap.c 9:3-15	Ignore	v
	*	*	Stop	v

This window displays the rules that control how messages are reported in the **C-RUN Messages** window. When a potential error is detected, it is matched against these rules (from top to bottom) and the action taken is determined by the first rule that matches. At

the bottom, there is always a catch-all rule that matches all messages. This rule can be modified using **Default action** in the **C-RUN Messages** window.

* is used as a wildcard.

See also *Using C-RUN*, page 292.

Requirements

A license for the C-RUN product.

Display area

The display area provides information in these columns:

Check

The name of the runtime error that this rule matches.

Source File

The name of the source file and possibly the location in the file to match.

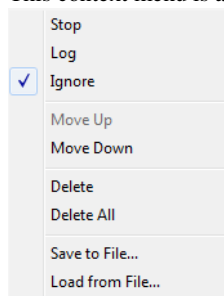
Action

The action to take for errors that match the rule:

- **Stop** stops the execution and logs the error
- **Log** logs the error but continues the execution
- **Ignore** neither logs nor stops.

Context menu

This context menu is available:



These commands are available:

Stop/Log/Ignore

Selects the action to take when a message matches the selected rule.

Move Up/Down

Moves the selected rule up/down one step.

Delete

Deletes the selected rule.

Delete All

Deletes all rules.

Save to File

Opens a dialog box where you can choose to save rules, see **Load from File**.
See also `--rtc_rules`, page 328.

Load from File

Opens a dialog box where you can choose to load rules from a file.

Compiler and linker reference for C-RUN

Reference information about:

- `--bounds_table_size`, page 321 (linker option)
- `--debug_heap`, page 321 (linker option)
- `--generate_entries_without_bounds`, page 321 (compiler option)
- `--ignore_uninstrumented_pointers`, page 322 (compiler option)
- `--ignore_uninstrumented_pointers`, page 322 (linker option)
- `--runtime_checking`, page 322 (compiler option)
- `#pragma default_no_bounds`, page 323
- `#pragma define_with_bounds`, page 324
- `#pragma define_without_bounds`, page 324
- `#pragma disable_check`, page 324
- `#pragma generate_entry_without_bounds`, page 325
- `#pragma no_arith_checks`, page 325
- `#pragma no_bounds`, page 325
- `__as_get_base`, page 326
- `__as_get_bound`, page 326
- `__as_make_bounds`, page 326

--bounds_table_size

Syntax	<code>--bounds_table_size records[:buckets] (bytes)</code>	
Parameters	<i>records</i>	The number of records.
	<i>:buckets</i>	The number of buckets.
	<i>(bytes)</i>	The number of bytes, within parentheses.
For use with	The linker.	
Description	<p>Use this linker option to specify the size of the global bounds table, which is used for tracking the bounds of pointers in memory.</p> <p>You can specify the number of records in the table (the number of pointers it can keep bounds for). If you do, you can also specify the number of buckets (a power of two), which will affect the speed of lookups. If not specified, the number of buckets is a power of two that is at least 6 times the number of records.</p> <p>Alternatively, you can specify the total number of bytes to use for records and buckets.</p>	
See also	<i>Detecting accesses outside the bounds of arrays and other objects</i> , page 301.	



Project>Options>Runtime Checking>Number of entries

--debug_heap


Syntax	<code>--debug_heap</code>	
For use with	The linker.	
Description	Use this linker option to use the checked heap.	
See also	<i>The checked heap provided by the library</i> , page 291.	




Project>Options>Runtime Checking>Use checked heap

--generate_entries_without_bounds


Syntax	<code>--generate_entries_without_bounds</code>
--------	--

For use with	The compiler.
Description	Use this compiler option to generate extra functions for use from non-instrumented code. This option requires that out-of-bounds checking is enabled.
See also	<i>Detecting accesses outside the bounds of arrays and other objects</i> , page 301.
	 Project>Options>Runtime Checking>Generate functions callable from non-instrumented code

--ignore_uninstrumented_pointers

Syntax	<code>--ignore_uninstrumented_pointers</code>
For use with	The compiler.
Description	Use this compiler option to disable checking of accesses via pointers from non-instrumented functions.
See also	<i>Detecting accesses outside the bounds of arrays and other objects</i> , page 301.
	 Project>Options>Runtime Checking>Check pointers from non-instrumented functions

--ignore_uninstrumented_pointers

Syntax	<code>--ignore_uninstrumented_pointers</code>
For use with	The linker.
Description	Use this linker option to disable checking of accessing via pointers in memory for which no bounds have been set.
See also	<i>Detecting accesses outside the bounds of arrays and other objects</i> , page 301.
	 Project>Options>Runtime Checking>Check pointers from non-instrumented memory

--runtime_checking

Syntax	<code>--runtime_checking param ,param, ...</code>
--------	---

Parameters	<i>param</i> is one of:	
	<code>signed_overflow unsigned_overflow</code>	Checks for signed or unsigned overflow in integer operations.
	<code>integer_conversion implicit_integer_conversion</code>	Checks for implicit or explicit integer conversions resulting in a change of value.
	<code>div_by_zero</code>	Checks for division by zero.
	<code>signed_shift unsigned_shift</code>	Checks for bit loss or implementation-dependent results when shifting.
	<code>switch</code>	Checks for unhandled cases in <code>switch</code> statements.
	<code>bounds</code>	Checks for accesses outside the bounds of arrays and other objects.
	<code>bounds_no_checks</code>	Tracks pointer bounds, but performs no checks. See also <code>#pragma disable_check = bounds</code> .

For use with The compiler.

Description Use this compiler option to enable runtime error checking.

See also *Introduction to runtime error checking*, page 289.



To set related options, choose:

Project>Options>Runtime Checking

#pragma default_no_bounds

Syntax	<code>#pragma default_no_bounds [=on off]</code>	
Parameters	<code>on</code>	Makes the default for all functions declared from this point be as if they were declared with <code>#pragma no_bounds</code> .
	<code>off</code>	Turns off the default.
Description	Use this pragma directive to apply <code>#pragma no_bounds</code> to a whole set of functions, for example around a header file declaring the interface to unchecked code.	

See also

Detecting accesses outside the bounds of arrays and other objects, page 301.**#pragma define_with_bounds**

Syntax	<code>#pragma define_with_bounds</code>
Description	<p>You can only use this pragma directive on a function that is declared with <code>#pragma no_bounds</code> (or equivalent). The function will then be instrumented to track pointer bounds, but not to perform any bounds checks. Any calls to the function will be to the version without extra bounds information.</p> <p>This is useful for writing a checking version of a function based on the non-checking version.</p>

#pragma define_without_bounds


Syntax	<code>#pragma define_without_bounds</code>
Description	<p>Use this pragma directive to define the version of a function that does not have extra bounds information. The code of the function is still instrumented to track pointer bounds (and checks are also inserted, unless <code>#pragma disable_check = bounds</code> is used).</p> <p>This can be useful for functions that are exclusively called from code that does not track pointer bounds, and where the bounds can be inferred from other arguments, or in some other way.</p>
Example	<pre>/* p points to an array of n integers */ void fun(int * p, int n) { /* Set up bounds for p. */ p = __as_make_bounds(p, n); ... }</pre>

#pragma disable_check

Syntax	<code>#pragma disable_check = bounds</code>		
Parameters	<table> <tr> <td><code>bounds</code></td><td>Does not check accesses against bounds.</td></tr> </table>	<code>bounds</code>	Does not check accesses against bounds.
<code>bounds</code>	Does not check accesses against bounds.		

Description	Use this pragma directive to specify that the immediately following function does not check accesses against bounds. If compiled with bounds checking, the function will be instrumented to track bounds, but will perform no checks.
-------------	---

#pragma generate_entry_without_bounds

Syntax	<code>#pragma generate_entry_without_bounds</code>
Description	<p>Use this pragma directive to enable generation of an extra entry without bounds for the immediately following function. This extra entry (function) can be called from code which is not instrumented for bounds checking. It takes no extra hidden parameters, and does not add any information about bounds for returned pointers. Any pointers passed into such a function are given bounds that will cause an error for any access. If you use <code>--ignore_uninstrumented_pointers</code>, the given bounds will not cause errors.</p> <p> It is an error to use this pragma directive on a function where no such entry can be generated. This includes functions that take a variable number of arguments, and functions that take one or more function pointers to functions that take or return values that contain pointers.</p> <p>It is not an error to use this pragma directive on a function that does not need such an entry (because it takes no pointers, or because it is declared with <code>#pragma no_bounds</code>). In this case, no extra entry is generated.</p>
See also	<i>Detecting accesses outside the bounds of arrays and other objects</i> , page 301.

#pragma no_arith_checks

Syntax	<code>#pragma no_arith_checks</code>
Description	Use this pragma directive to specify that no C-RUN arithmetic checks will be performed in the function that follows.

#pragma no_bounds

Syntax	<code>#pragma no_bounds</code>
Description	Use this pragma directive to specify that the immediately following function is not instrumented for bounds checking. No extra hidden bounds parameters will be passed when this function is called, and it will not return bounds for pointers, if any, in its return value.

See also *Detecting accesses outside the bounds of arrays and other objects*, page 301.

__as_get_base

Syntax	<code>__as_get_base(ptr)</code>	
Parameters	<i>ptr</i>	A pointer.
Description	Use this operator to create a pointer of the same type as <i>ptr</i> , representing the base of the area pointed to by <i>ptr</i> .	
Example	<code>base = __as_get_base(my_ptr);</code>	

__as_get_bound

Syntax	<code>__as_get_bound(ptr)</code>	
Parameters	<i>ptr</i>	A pointer.
Description	Use this operator to create a pointer of the same type as <i>ptr</i> , representing the upper bound of the area pointed to by <i>ptr</i> .	
Example	<code>bound = __as_get_bound(my_ptr);</code>	

__as_make_bounds

Syntax	<code>__as_make_bounds(ptr, number)</code> <code>__as_make_bounds(ptr, base, bound)</code>	
Parameters	<i>ptr</i>	A pointer that has no bounds.
	<i>number</i>	The number of elements.
	<i>base</i>	The start of the object pointed to.
	<i>bound</i>	The end of the object pointed to.
Description	Use this operator to create a pointer with bounds information. Use the first syntax to create the bounds <i>ptr</i> up to <i>ptr + size</i> for <i>ptr</i> . The second syntax has explicit bounds.	

base is a pointer to the first element of the area. *bound* is a pointer to just beyond the area. Except that each expression will be evaluated only once, the two-parameter variant is equivalent to `__as_make_bounds(ptr, ptr, ptr + size)`.

Example

```
/* Starting here, p points to a single element */
p = __as_make_bounds(p, 1);
/* Call fun with a pointer with the specified bounds */
fun(__as_make_bounds(q, start, end));
```

cspybat options for C-RUN

Reference information about:

- `--rtc_enable`, page 327
- `--rtc_output`, page 327
- `--rtc_raw_to_txt`, page 328
- `--rtc_rules`, page 328

--rtc_enable

Syntax

```
--rtc_enable
```

Note that this option must be placed before the `--backend` option on the command line.

For use with

```
cspybat
```

Description

Use this option to enable C-RUN run-time checking in `cspybat`. This option is automatically enabled if any of the other `-rtc_*` options are used.



This option is not available in the IDE.

--rtc_output

Syntax

```
--rtc_output file
```


Note that this option must be placed before the `--backend` option on the command line.

Parameters


file The file for output messages.

For use with


```
cspybat
```

Description	Use this option to specify to <code>cspybat</code> a file for the C-RUN message output, in text (filename extension <code>txt</code>) or XML (filename extension <code>xml</code>) format.
	 This option is not available in the IDE.

--rtc_raw_to_txt

Syntax	<code>--rtc_raw_to_txt=file</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
For use with	<code>cspybat</code>
Description	Use this option to make <code>cspybat</code> act as a runtime checking messages filter. The option reads a file and transforms each message into a properly formatted message (as in the C-RUN Messages window). The only limitation is that call stack information cannot be provided.
	 This option is not available in the IDE.

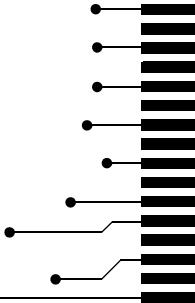
--rtc_rules

Syntax	<code>--rtc_rules file</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<div><div><code>file</code></div><div>The rules input file.</div></div>
For use with	<code>cspybat</code>
Description	Use this option to specify the name of the C-RUN rules file to <code>cspybat</code> .
See also	<i>C-RUN Messages Rules window</i> , page 318 for information about Save to File .
	 This option is not available in the IDE.

Part 3. Advanced debugging

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

- Interrupts
- C-SPY macros
- The C-SPY command line utility—`cspybat`





Interrupts

- Introduction to interrupts
- Using the interrupt system
- Reference information on interrupts

Introduction to interrupts

These topics are covered:

- Briefly about the interrupt simulation system
- Interrupt characteristics
- 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 369
- *Breakpoints*, page 123
- *The IAR C/C++ Development Guide for RX*

BRIEFLY ABOUT THE INTERRUPT SIMULATION SYSTEM

By simulating interrupts, you can test the logic of your interrupt service routines and debug the interrupt handling in the target system long before any hardware is available. If you use simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices.

The C-SPY Simulator includes an interrupt simulation system where you can simulate the execution of interrupts during debugging. You can configure the interrupt simulation system so that it resembles your hardware interrupt system.

The interrupt system has the following features:

- Simulated interrupt support for the RX microcontroller
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for various devices

- Configuration of hold time, probability, and timing variation
- State information for locating timing problems
- Configuration of interrupts using a dialog box or a C-SPY system macro—that is, one interactive and one automating interface. In addition, you can instantly force an interrupt.
- A log window that continuously displays events for each defined interrupt.
- A status window that shows the current interrupt activities.

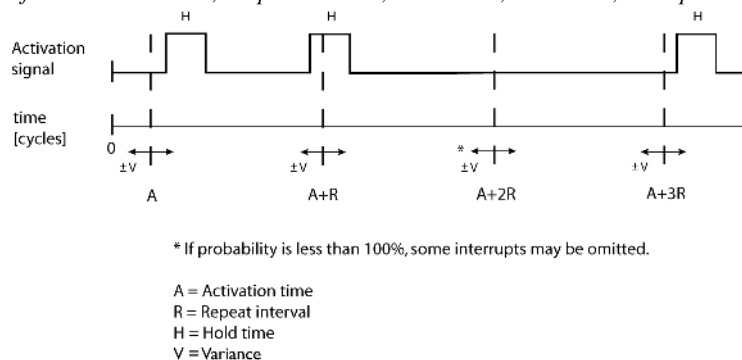
All interrupts you define using the **Interrupt Setup** dialog box are preserved between debug sessions, unless you remove them. A forced interrupt, on the other hand, exists only until it has been serviced and is not preserved between sessions.



The interrupt simulation system is activated by default, but if not required, you can turn off the interrupt simulation system to speed up the simulation. To turn it off, use either the **Interrupt Setup** dialog box or a system macro.

INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability*.



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.

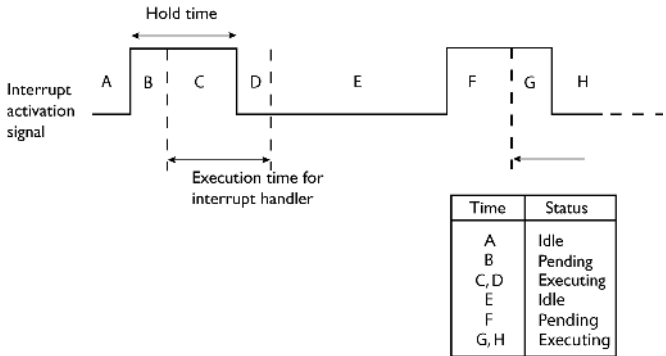
To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—

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

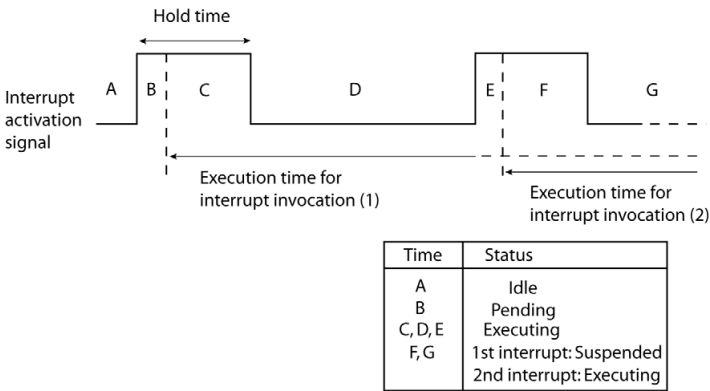
The interrupt simulation system contains status information that you can use for locating timing problems in your application. The **Interrupt Status** window displays the available status information. For an interrupt, these states can be displayed: *Idle*, *Pending*, *Executing*, or *Suspended*.

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:



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

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



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 Setup** dialog box.

For more information about each macro, see *Reference information on C-SPY system macros*, page 369.

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 47.

BRIEFLY ABOUT INTERRUPT LOGGING

Interrupt logging provides you with comprehensive information about the interrupt events. This might be useful, for example, to help you locate which interrupts you can fine-tune to become faster. You can log entrances and exits to and from interrupts. You can also log internal interrupt status information, such as triggered, expired, etc. 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 338.

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 357 for details about how to use a setup file to define simulated interrupts at C-SPY startup
- The tutorial *Simulating an interrupt* in the Information Center.

SIMULATING A SIMPLE INTERRUPT

This example demonstrates the method for simulating a timer interrupt. However, the procedure can also be used for other types of interrupts.

To simulate and debug an interrupt:

- 1 Assume this simple application which contains an interrupt service routine for a timer, which increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```
#include <sdtio.h>
#include "ior5f56108.h"
#include <intrinsics.h>

volatile int ticks = 0;
void main (void)
{
    /* Add your timer setup code here */

    __enable_interrupt();          /* Enable interrupts */

    while (ticks < 100);          /* Endless loop */
    printf("Done\n");
}

/* Timer interrupt service routine */
#pragma vector = INT_TIMER
__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** to see the name and location of the device description file that is being used. Open it to define an interrupt that C-SPY can simulate. The device description file contains the information you need.
- 4 Build your project and start the simulator.

- 5 Choose **Simulator>Interrupt Setup** to open the **Interrupt Setup** dialog box. Select the **Enable interrupt simulation** option to enable interrupt simulation. Click **New** to open the **Edit Interrupt** dialog box. For the timer example, verify these settings:

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

Table 14: Timer interrupt settings

Click **OK**.

- 6 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:
 - Generate an interrupt when the cycle counter has passed 4000
 - Continuously repeat the interrupt after approximately 2000 cycles.
- 7 To watch the interrupt in action, choose **Simulator>Interrupt Log** to open the Interrupt Log window.
- 8 From the context menu, available in the Interrupt Log window, choose **Enable** to enable the logging. If you restart program execution, status information about entrances and exits to and from interrupts will now appear in the Interrupt Log window.

For information about how to get a graphical representation of the interrupts correlated with a time axis, see Timeline window—Interrupt Log graph.

SIMULATING AN INTERRUPT IN A MULTI-TASK SYSTEM

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

To simulate a normal interrupt exit:

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

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

GETTING STARTED USING INTERRUPT LOGGING

- 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.

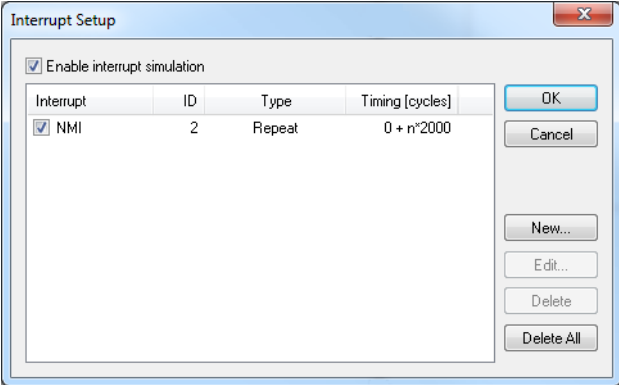
Reference information on interrupts

Reference information about:

- *Interrupt Setup dialog box*, page 339
- *Edit Interrupt dialog box*, page 341
- *Forced Interrupt window*, page 342
- *Interrupt Status window*, page 343
- *Interrupt Log window*, page 345
- *Interrupt Log Summary window*, page 348
- *Timeline window—Interrupt Log graph*, page 350

Interrupt Setup dialog box

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



This dialog box lists all defined interrupts. Use this dialog box to enable or disable the interrupt simulation system, as well as to enable or disable individual interrupts.

See also *Using the interrupt system*, page 335.

Requirements

The C-SPY simulator.

Enable interrupt simulation

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

Display area

This area contains these columns:

Interrupt

Lists all interrupts. Use the checkbox to enable or disable the interrupt.

ID

A unique interrupt identifier.

Type

Shows the type of the interrupt. The type can be one of:

Forced, a single-occasion interrupt defined in the **Forced Interrupt** 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)**.

Timing

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

Buttons

These buttons are available:

New

Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 341.

Edit

Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 341.

Delete

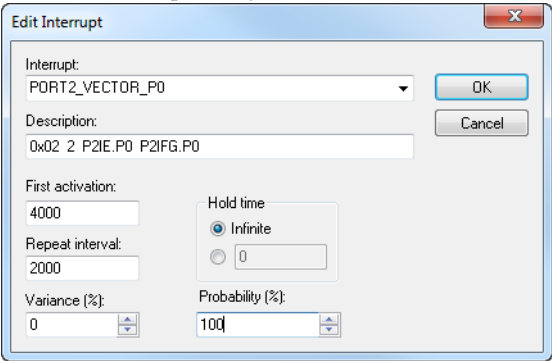
Removes the selected interrupt.

Delete All

Removes all interrupts.

Edit Interrupt dialog box

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



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

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

See also *Using the interrupt system*, page 335.

Requirements

The C-SPY simulator.

Interrupt

Selects the interrupt that you want to edit. The drop-down list contains all available interrupts. Your selection will automatically update the **Description** box. The list is populated with entries from the device description file that you have selected.

Description

A description of the selected interrupt, if available. The description is retrieved from the selected device description file. See this file for a detailed description. For interrupts specified using the system macro `__orderInterrupt`, the **Description** box is empty.

First activation

Specify the value of the cycle counter after which the specified type of interrupt will be generated.

Repeat interval

Specify the periodicity of the interrupt in cycles.

Variance %

Selects a timing variation range, as a percentage of the repeat interval, in which the interrupt might occur for a period. For example, if the repeat interval is 100 and the variance 5%, the interrupt might occur anywhere between T=95 and T=105, to simulate a variation in the timing.

Hold time

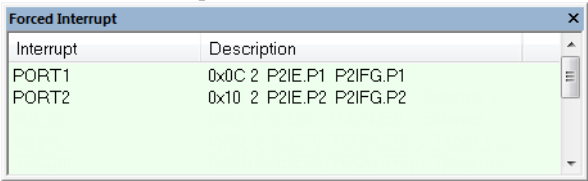
Specify how long, in cycles, the interrupt remains pending until removed if it has not been processed. If you select **Infinite**, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

Probability %

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

Forced Interrupt window

The **Forced Interrupt** window is available from the C-SPY driver menu.



Interrupt	Description
PORT1	0x0C 2 P2IE.P1 P2IFG.P1
PORT2	0x10 2 P2IE.P2 P2IFG.P2

Use this window to force 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 Setup dialog box*, page 339.
- 2 Double-click the interrupt in the **Forced Interrupt** window, or activate it by using the **Force** command available on the context menu.

Requirements

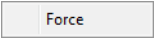
The C-SPY simulator.

Display area

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

Context menu

This context menu is available:



This command is available:

Force

Triggers the interrupt you selected in the display area.

Interrupt Status window

The **Interrupt Status** window is available from the C-SPY driver menu.

Interrupt Status					
Interrupt	ID	Type	Status	Next Time	Timing [cycles]
TIM_INT	1	Single	Idle	0	0
NMI	0	Single	Idle	0	0
SCIO_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 **Forced Interrupt** 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	IROT1	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 338.

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 350.

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 **Forced Interrupt** window.

Enter, the interrupt is currently executing.

Leave, the interrupt has been executed.

Expired, the interrupt hold time has expired without the interrupt being executed.

Rejected, the interrupt has been rejected because the necessary interrupt registers were not set up to accept the interrupt.

Program Counter

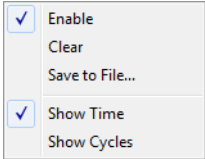
The value of the program counter when the event occurred.

Execution Time/Cycles

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

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 Log Summary								
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 338.

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 350.

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:

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

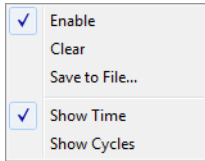
The longest time between two interrupts of this type.

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

** 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

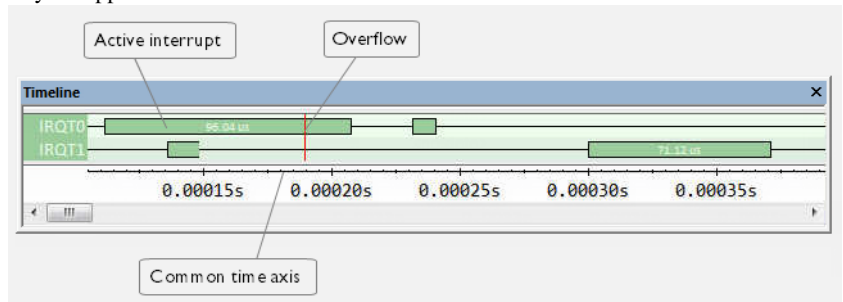
Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.

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

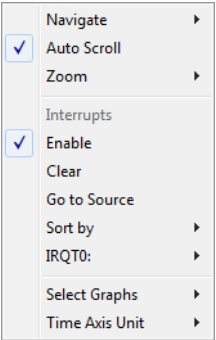
- The label area at the left end of the graph displays the names of the interrupts.
- The graph itself shows active interrupts as a thick green horizontal bar where the white figure indicates the time spent in the interrupt. This graph is a graphical representation of the information in the **Interrupt Log** window, see *Interrupt Log window*, page 345.
- 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 367.

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 362.

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 54

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 359.
- You can register macros interactively in the **Macro Registration** window, see *Macro Registration window*, page 408. Registered macros appear in the **Debugger Macros** window, see *Debugger Macros window*, page 410.
- 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 390.

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 359.
- 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 359.
- 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 412.
- 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 360.

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

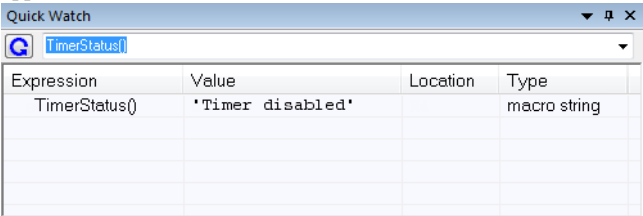
The **Quick Watch** window lets you dynamically choose when to execute a macro function.

- 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 screenshot shows the 'Quick Watch' window in a debugger. At the top, there is a search bar containing 'TimerStatus()'. Below it is a table with four columns: 'Expression', 'Value', 'Location', and 'Type'. The first row of the table contains the following data: 'TimerStatus()' in the Expression column, ''Timer disabled'' in the Value column, an empty cell in the Location column, and 'macro string' in the Type column. There are several empty rows below the first one.

Expression	Value	Location	Type
TimerStatus()	'Timer disabled'		macro string

The macro will automatically be displayed in the **Quick Watch** window. For more information, see *Quick Watch window*, page 116.

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 141
- Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 133.

- 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 365.

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 362
- *Macro variables*, page 362
- *Macro parameters*, page 363
- *Macro strings*, page 363
- *Macro statements*, page 364
- *Formatted output*, page 365

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 98.

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 double, value 3.5.
<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 15: 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 432.

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 365.

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 98.

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 argList;</code>	Prints the output to the Debug Log window.
<code>__fmessage file, argList;</code>	Prints the output to the designated file.
<code>__smessage argList;</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 385.

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.

Another example:

```
__message "The character '", cvar:%c, "' has the decimal value
", cvar;
```

Depending on the value of the variables, this produces this message:

The character 'A' has the decimal value 65

Note: A character enclosed in single quotes (a character literal) is an integer constant and is not automatically formatted as a character. For example:

```
__message 'A', " is the numeric value of the character ",
'A':%c;
```

would produce:

65 is the numeric value of the character A

Note: The default format for certain types is primarily designed to be useful in the **Watch** window and other related windows. For example, a value of type `char` is formatted as 'A' (0x41), while a pointer to a character (potentially a C string) is formatted as 0x8102 "Hello", where the string part shows the beginning of the string (currently up to 60 characters).

When printing a value of type `char*`, use the `%x` format specifier to print just the pointer value in hexadecimal notation, or use the system macro `__toString` to get the full string value.

Reference information on reserved setup macro function names

There are reserved setup macro function names that you can use for defining your setup macro functions. By using these reserved names, your function will be executed at defined stages during execution. For more information, see *Briefly about setup macro functions and files*, page 356.

Reference information about:

- `execUserAttach`
- `execUserPreload`
- `execUserSetup`
- `execUserPreReset`
- `execUserReset`
- `execUserExit`

execUserAttach

Syntax	<code>execUserAttach</code>
For use with	All C-SPY drivers.
Description	<p>Called after the debugger attaches to a running application at its current location without resetting the target system (the option Attach to running target).</p> <p>Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.</p>

execUserPreload

Syntax	<code>execUserPreload</code>
For use with	All C-SPY drivers.
Description	<p>Called after communication with the target system is established but before downloading the target application.</p> <p>Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.</p>

execUserSetup

Syntax	<code>execUserSetup</code>
For use with	All C-SPY drivers.
Description	<p>Called once after the target application is downloaded.</p> <p>Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.</p>



If you define interrupts or breakpoints in a macro file that is executed at system start (using `execUserSetup`) we strongly recommend that you also make sure that they are removed at system shutdown (using `execUserExit`). An example is available in `SetupSimple.mac`, see the tutorials in the Information Center.

The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time `execUserSetup` is executed again. This seriously affects the execution speed.

execUserPreReset

Syntax	<code>execUserPreReset</code>
For use with	All C-SPY drivers.
Description	<p>Called each time just before the reset command is issued.</p> <p>Implement this macro to set up any required device state.</p>

execUserReset

Syntax	execUserReset
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	execUserExit
For use with	All C-SPY drivers.
Description	Called once when the debug session ends. Implement this macro to save status data etc.

Reference information on C-SPY system macros

This section gives reference information about each of the C-SPY system macros.
This table summarizes the pre-defined system macros:

Macro	Description
__abortLaunch	Aborts the launch of the debugger
__cancelAllInterrupts	Cancels all ordered interrupts
__cancelInterrupt	Cancels an interrupt
__clearBreak	Clears a breakpoint
__closeFile	Closes a file that was opened by __openFile
__delay	Delays execution
__disableInterrupts	Disables generation of interrupts
__driverType	Verifies the driver type
__enableInterrupts	Enables generation of interrupts
__evaluate	Interprets the input string as an expression and evaluates it
__fillMemory8	Fills a specified memory area with a byte value

Table 16: Summary of system macros

Macro	Description
__fillMemory16	Fills a specified memory area with a 2-byte value
__fillMemory32	Fills a specified memory area with a 4-byte value
__getNumberOfCores	Gets the number local cores being debugged. Only useful for IAR Embedded Workbench products that support multicore debugging.
__getSelectedCore	Only for use with IAR Embedded Workbench products that support multicore debugging
__isBatchMode	Checks if C-SPY is running in batch mode or not.
__isMacroSymbolDefined	Checks if a C-SPY macro symbol is defined.
__loadImage	Loads a debug image
__memoryRestore	Restores the contents of a file to a specified memory zone
__memorySave	Saves the contents of a specified memory area to a file
__messageBoxYesCancel	Displays a Yes/Cancel dialog box for user interaction
__messageBoxYesNo	Displays a Yes/No dialog box for user interaction
__openFile	Opens a file for I/O operations
__orderInterrupt	Generates an interrupt
__popSimulatorInterruptExecutingStack	Informs the interrupt simulation system that an interrupt handler has finished executing
__readFile	Reads from the specified file
__readFileByte	Reads one byte from the specified file
__readMemory8, __readMemoryByte	Reads one byte from the specified memory location
__readMemory16	Reads two bytes from the specified memory location
__readMemory32	Reads four bytes from the specified memory location
__registerMacroFile	Registers macros from the specified file
__resetFile	Rewinds a file opened by __openFile
__selectCore	Only for use with IAR Embedded Workbench products that support multicore debugging
__setCodeBreak	Sets a code breakpoint
__setDataBreak	Sets a data breakpoint
__setDataLogBreak	Sets a data log breakpoint
__setLogBreak	Sets a log breakpoint

Table 16: Summary of system macros (Continued)

Macro	Description
<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
<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

Table 16: Summary of system macros (Continued)

__abortLaunch

Syntax	<code>__abortLaunch(<i>message</i>)</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	<pre>if (!__messageBoxYesCancel("Do you want to mass erase to unlock the device?", "Unlocking device")) { __abortLaunch("Unlock canceled. Debug session cannot continue."); }</pre>

__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	int 0
Unsuccessful	Non-zero error number

Table 17: `__cancelInterrupt` return values

For use with

The C-SPY Simulator.

Description

Cancels the specified interrupt.

`__clearBreak`

Syntax

`__clearBreak(break_id)`

Parameters

break_id
The value returned by any of the set breakpoint macros.

Return value

int 0

For use with

All C-SPY drivers.

Description

Clears a user-defined breakpoint.

See also

Breakpoints, page 123.

`__closeFile`

Syntax

`__closeFile(fileHandle)`

Parameters

fileHandle
A macro variable used as filehandle by the `__openFile` macro.

Return value

int 0

For use with

All C-SPY drivers.

Description

Closes a file previously opened by `__openFile`.

__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	<code>__disableInterrupts()</code>						
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td><code>int 0</code></td></tr><tr><td>Unsuccessful</td><td>Non-zero error number</td></tr></table>	Result	Value	Successful	<code>int 0</code>	Unsuccessful	Non-zero error number
Result	Value						
Successful	<code>int 0</code>						
Unsuccessful	Non-zero error number						
For use with	The C-SPY Simulator.						
Description	Disables the generation of interrupts.						

Table 18: __disableInterrupts return values

__driverType

Syntax	<code>__driverType(<i>driver_id</i>)</code>
Parameters	<i>driver_id</i> A string corresponding to the driver you want to check for. Choose one of these: "sim" corresponds to the simulator driver "emue20" corresponds to the C-SPY E1/E20 driver "e2lite" corresponds to the C-SPY E2/E2 Lite or EZ-CUBE2 emulator "jlink" corresponds to the C-SPY J-Link driver

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 19: `__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 20: `__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 21: `__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

```
__fillMemory8(value, address, zone, length, format)
```

Parameters

value

An integer that specifies the value.

address

An integer that specifies the memory start address.

zone

A string that specifies the memory zone, see *C-SPY memory zones*, page 153.

length

An integer that specifies how many bytes are 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 a byte value.	
Example	<code>__fillMemory8(0x80, 0x700, "", 0x10, "OR");</code>	

__fillMemory16

Syntax	<code>__fillMemory16(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>	
Parameters	<i>value</i>	An integer that specifies the value.
	<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 153.
	<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	<pre>__fillMemory16(0xCDCD, 0x7000, "", 0x200, "Copy");</pre>
__fillMemory32	
Syntax	<pre>__fillMemory32(value, address, zone, length, format)</pre>
Parameters	<div><div><i>value</i></div><div>An integer that specifies the value.</div></div> <div><div><i>address</i></div><div>An integer that specifies the memory start address.</div></div> <div><div><i>zone</i></div><div>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 153.</div></div> <div><div><i>length</i></div><div>An integer that defines how many 4-byte entities to be affected.</div></div> <div><div><i>format</i></div><div>A string that specifies the exact fill operation to perform. Choose between:<div><div><div>Copy</div><div><i>value</i> will be copied to the specified memory area.</div></div><div><div>AND</div><div>An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</div></div><div><div>OR</div><div>An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</div></div><div><div>XOR</div><div>An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</div></div></div></div></div>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Fills a specified memory area with a 4-byte value.
Example	<pre>__fillMemory32(0x0000FFFF, 0x4000, "", 0x1000, "XOR");</pre>

__getNumberOfCores

Syntax	__getNumberOfCores()
Return value	The number of local cores being debugged.
For use with	
Description	This macro returns the number of local cores being debugged. It is only useful for IAR Embedded Workbench products that support multicore debugging.
Example	<pre>test () { __var i; for (i = 0; i < __getNumberOfCores(); i++) { __selectCore(i); __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n"; } }</pre>
See also	__getSelectedCore, page 379 and __selectCore, page 391

__getSelectedCore

Description	This macro returns 0 for a single-core system. It is only useful for IAR Embedded Workbench products that support multicore debugging
-------------	---

__isBatchMode

Syntax	__isBatchMode()						
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>True</td><td>int 1</td></tr><tr><td>False</td><td>int 0</td></tr></table>	Result	Value	True	int 1	False	int 0
Result	Value						
True	int 1						
False	int 0						
For use with	All C-SPY drivers.						

Table 22: __isBatchMode return values

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

__isMacroSymbolDefined

Syntax	<code>__isMacroSymbolDefined(<i>symbol</i>)</code>
Parameters	<i>symbol</i> The name of a C-SPY macro variable or macro function (a string).
Return value	1 if <i>symbol</i> is an existing macro symbol. 0 if <i>symbol</i> 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	<i>path</i> 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 RX</i> . <i>offset</i> An integer that identifies the offset to the destination address for the downloaded debug image.

debugInfoOnly

A non-zero integer value if no code or data should be downloaded to the target system, which means that C-SPY will only read the debug information from the debug file. Or, 0 (zero) for download.

Return value

Value	Result
Non-zero integer number	A unique module identification.
int 0	Loading failed.

Table 23: *__loadImage* return values

For use with

All C-SPY drivers.

Description

Loads a debug image (debug file).

Note: Images are only downloaded to RAM and no flash loading will be performed.

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 441 and *Loading multiple debug images*, page 49.

__memoryRestore

Syntax	<code>__memoryRestore(zone, filename, offset)</code>
Parameters	<p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 153.</p> <p><i>filename</i></p> <p>A string that specifies the file to be read. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide for RX</i>.</p> <p><i>offset</i></p> <p>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.</p>
Return value	<code>int 0</code>
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(" ", "c:\\temp\\saved_mem.hex", 0x400);</code>
See also	<i>Memory Restore dialog box</i> , page 164.

__memorySave

Syntax	<code>__memorySave(start, stop, format, filename, zerostart)</code>
Parameters	<p><i>start</i></p> <p>A string that specifies the first location of the memory area to be saved.</p> <p><i>stop</i></p> <p>A string that specifies the last location of the memory area to be saved.</p>

	<p><i>format</i></p> <p>A string that specifies the format to be used for the saved memory. Choose between:</p> <p>intel-extended</p> <p>motorola</p> <p>motorola-s19</p> <p>motorola-s28</p> <p>motorola-s37</p> <p><i>filename</i></p> <p>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 RX</i>.</p> <p><i>zerostart</i></p> <p>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.</p>
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(":0x00", ":0xFF", "intel-extended", "c:\\temp\\saved_memory.hex", 0);</pre>
See also	<i>Memory Save dialog box</i> , page 163.

__messageBoxYesCancel

Syntax	<code>__messageBoxYesCancel(message, caption)</code>
Parameters	<p><i>message</i></p> <p>A message that will appear in the message box.</p>

	<i>caption</i> The title that will appear in the message box.						
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Yes</td><td>1</td></tr><tr><td>No</td><td>0</td></tr></table> <i>Table 24: __messageBoxYesCancel return values</i>	Result	Value	Yes	1	No	0
Result	Value						
Yes	1						
No	0						
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	<code>__messageBoxYesNo (message, caption)</code>						
Parameters	<i>message</i> A message that will appear in the message box. <i>caption</i> The title that will appear in the message box.						
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Yes</td><td>1</td></tr><tr><td>No</td><td>0</td></tr></table> <i>Table 25: __messageBoxYesNo return values</i>	Result	Value	Yes	1	No	0
Result	Value						
Yes	1						
No	0						
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 RX*.

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 26: __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 RX*.

__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.

first_activation

The first activation time in cycles (integer)

repeat_interval

The periodicity in cycles (integer)

variance

The timing variation range in percent (integer between 0 and 100)

infinite_hold_time

1 if infinite, otherwise 0.

hold_time

The hold time (integer)

probability

The probability in percent (integer between 0 and 100)

Return value

The macro returns an interrupt identifier (unsigned long).

If the syntax of *specification* is incorrect, it returns -1.

For use with	The C-SPY simulator.
Description	Generates an interrupt.
Example	<p>This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles:</p> <pre>__orderInterrupt("SCIO_RXI0", 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 task-switching. In this case, the interrupt simulation system cannot automatically detect that the interrupt has finished executing.</p>
See also	<i>Simulating an interrupt in a multi-task system</i> , page 337.

__readFile

Syntax	<code>__readFile(fileHandle, valuePtr)</code>						
Parameters	<p><i>fileHandle</i></p> <p>A macro variable used as filehandle by the __openFile macro.</p> <p><i>valuePtr</i></p> <p>A pointer to a variable.</p>						
Return value	<table><tr><th>Result</th><th>Value</th></tr><tr><td>Successful</td><td>0</td></tr><tr><td>Unsuccessful</td><td>Non-zero error number</td></tr></table>	Result	Value	Successful	0	Unsuccessful	Non-zero error number
Result	Value						
Successful	0						
Unsuccessful	Non-zero error number						

Table 27: __readFile return values

For use with	All C-SPY drivers.
Description	<p>Reads a sequence of hexadecimal digits from the given file and converts them to an unsigned long which is assigned to the <i>value</i> parameter, which should be a pointer to a macro variable.</p> <p>Only printable characters representing hexadecimal digits and white-space characters are accepted, no other characters are allowed.</p>
Example	<pre>__var number; if (__readFile(myFileHandle, &number) == 0) { // Do something with number }</pre> <p>In this example, if the file pointed to by <code>myFileHandle</code> contains the ASCII characters 1234 abcd 90ef, consecutive reads will assign the values 0x1234 0xabcd 0x90ef to the variable <code>number</code>.</p>

__readFileByte

Syntax	<code>__readFileByte(<i>fileHandle</i>)</code>
Parameters	<p><i>fileHandle</i></p> <p>A macro variable used as filehandle by the <code>__openFile</code> macro.</p>
Return value	-1 upon error or end-of-file, otherwise a value between 0 and 255.
For use with	All C-SPY drivers.
Description	Reads one byte from a file.
Example	<pre>__var byte; while ((byte = __readFileByte(myFileHandle)) != -1) { /* Do something with byte */ }</pre>

__readMemory8, __readMemoryByte

Syntax	<pre>__readMemory8(<i>address</i>, <i>zone</i>) __readMemoryByte(<i>address</i>, <i>zone</i>)</pre>
--------	---

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 153.</p>
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, "");</code>

__readMemory16

Syntax	<code>__readMemory16(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 153.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads a two-byte word from a given memory location.
Example	<code>__readMemory16(0x0108, "");</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 153.</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	<pre>__readMemory32(0x0108, "");</pre>

__registerMacroFile

Syntax	<pre>__registerMacroFile(filename)</pre>
Parameters	<p><i>filename</i></p> <p>A file containing the macros to be registered (string). The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide for RX</i>.</p>
Return value	int 0
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	<pre>__registerMacroFile("c:\\testdir\\macro.mac");</pre>
See also	<i>Using C-SPY macros</i> , page 357.

__resetFile

Syntax	<pre>__resetFile(fileHandle)</pre>
Parameters	<p><i>fileHandle</i></p> <p>A macro variable used as filehandle by the <code>__openFile</code> macro.</p>
Return value	int 0
For use with	All C-SPY drivers.
Description	Rewinds a file previously opened by <code>__openFile</code> .

__selectCore

Description

This macro can only be used with IAR Embedded Workbench products that support multicore debugging.

__setCodeBreak

Syntax

```
__setCodeBreak(location, count, condition, cond_type, action)
```

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 148.

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. 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 28: __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 123.

__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 148.
- 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.

Table 29: __setDataBreak return values

	Result	Value
	Unsuccessful	0

Table 29: `__setDataBreak` return values (Continued)

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	<pre>__var brk; brk = __setDataBreak(":0x4710", 3, "d>6", "TRUE", "W", "ActionData()"); ... __clearBreak(brk);</pre>
See also	<i>Breakpoints</i> , page 123.

`__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, 16, and 32-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><tr><th>Result</th><th>Value</th></tr><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></table>	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						

Table 30: `__setDataLogBreak` return values

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

```
__var brk;
brk = __setDataLogBreak("MyVar", "R");
...
__clearBreak(brk);
```

See also *Breakpoints*, page 123 and *Getting started using data logging*, page 215.

__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 148.
- 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 31: __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 365 and *Breakpoints*, page 123.

__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 148.

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 32: `__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 148.

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 33: `__setTraceStartBreak` return values

For use with

The C-SPY simulator.

Description

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

Example

```
__var startTraceBp;
__var stopTraceBp;

traceOn()
{
    startTraceBp = __setTraceStartBreak
        ("C:\\TEMP\\Utilities.c).23.1");
    stopTraceBp = __setTraceStopBreak
        ("C:\\temp\\Utilities.c).30.1");
}

traceOff()
{
    __clearBreak(startTraceBp);
    __clearBreak(stopTraceBp);
}
```

See also *Trace Start Trigger breakpoint dialog box*, page 205.

__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 148.

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 34: __setTraceStopBreak return values

For use with

The C-SPY simulator.

Description

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

Example

See *__setTraceStartBreak*, page 396.

See also *Trace Stop Trigger breakpoint dialog box*, page 206.

__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 35: __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 363.

__subString

Syntax	<code>__subString(<i>macroString</i>, <i>position</i>, <i>length</i>)</code>
Parameters	<p><i>macroString</i></p> <p>A macro string.</p> <p><i>position</i></p> <p>The start position of the substring. The first position is 0.</p> <p><i>length</i></p> <p>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 363.

__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	<pre>__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;</pre>
---------	---

__targetDebuggerVersion

Syntax	<code>__targetDebuggerVersion()</code>
Return value	A string that represents the version number of the C-SPY debugger processor module.
For use with	All C-SPY drivers.
Description	This macro returns the version number of the C-SPY debugger processor module.
Example	<pre>__var toolVer; toolVer = __targetDebuggerVersion(); __message "The target debugger version is, ", toolVer;</pre>

__toLower

Syntax	<code>__toLower(<i>macroString</i>)</code>
Parameters	<i>macroString</i> 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 <i>macroString</i> where all the characters have been converted to lower case.
Example	<pre>__toLower("IAR")</pre> <p>The resulting macro string contains <i>iar</i>.</p>

```
__toLower("Mix42")
```

The resulting macro string contains `mix42`.

See also

Macro strings, page 363.

__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 363.

__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	<pre>__toUpper("string")</pre> <p>The resulting macro string contains <i>STRING</i>.</p>
See also	<i>Macro strings</i> , page 363.

__unloadImage

Syntax	<pre>__unloadImage(module_id)</pre>
Parameters	<p><i>module_id</i></p> <p>An integer which represents a unique module identification, which is retrieved as a return value from the corresponding <code>__loadImage</code> C-SPY macro.</p>

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 36: `__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 49 and <i>Images</i> , page 441.

__wallTime_ms

Syntax	<pre>__wallTime_ms()</pre>
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 <i>fileHandle</i> .
-------------	---

__writeMemory8, __writeMemoryByte

Syntax	<code>__writeMemory8(value, address, zone)</code> <code>__writeMemoryByte(value, address, zone)</code>
Parameters	<i>value</i> An integer. <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 153.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes one byte to a given memory location.
Example	<code>__writeMemory8(0x2F, 0x8020, "");</code>

__writeMemory16

Syntax	<code>__writeMemory16(value, address, zone)</code>
Parameters	<i>value</i> An integer. <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 153.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes two bytes to a given memory location.

Example	<pre>__writeMemory16(0x2FFF, 0x8020, "");</pre>
__writeMemory32	
Syntax	<pre>__writeMemory32(value, address, zone)</pre>
Parameters	<p><i>value</i> An integer.</p> <p><i>address</i> The memory address (integer).</p> <p><i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 153.</p>
Return value	<pre>int 0</pre>
For use with	All C-SPY drivers.
Description	Writes four bytes to a given memory location.
Example	<pre>__writeMemory32(0x5555FFFF, 0x8020, "");</pre>

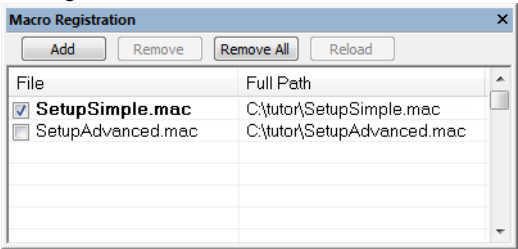
Graphical environment for macros

Reference information about:

- *Macro Registration window*, page 408
- *Debugger Macros window*, page 410
- *Macro Quicklaunch window*, page 412

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 358.

Requirements

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

Display area

This area contains these columns:

File

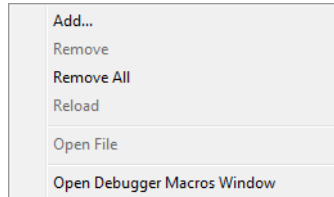
The name of an available macro file. To register the macro file, select the check box to the left of the filename. The name of a registered macro file appears in bold style.

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

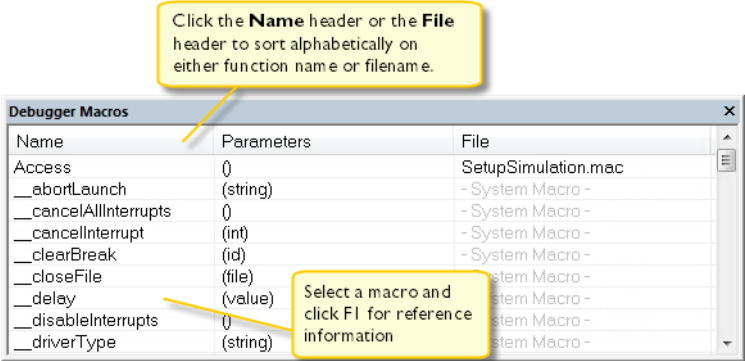
Opens the selected macro file in the editor window.

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:

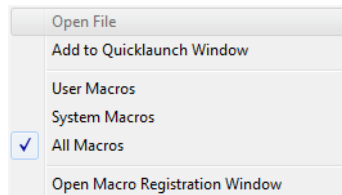
Name
The name of the debugger macro.

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

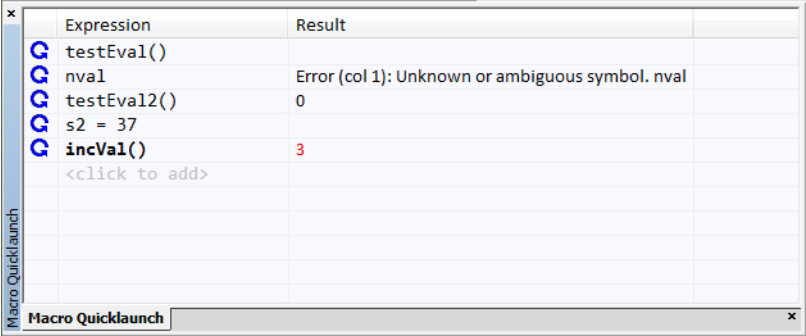
Lists all debugger macros, both predefined system macros and your own.

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 358.

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 358.

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

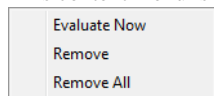
One or several expressions that you want to evaluate. Click <click to add> to add an expression. If the return value has changed since last time, the value will be displayed in red.

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:

- Before running `cspybat` for the first time
- Starting `cspybat`
- Output
- Invocation syntax

BEFORE RUNNING CSPYBAT FOR THE FIRST TIME

Before you run `cspybat` for the first time using a hardware debugger, you must:

- 1 Start the IAR Embedded Workbench IDE and set up the hardware debugger in the **Hardware Setup** dialog box—available from the **C-SPY driver** menu when you start a debug session. Save the project. The settings are saved to a file.
- 2 Set up the environment variable `CSPYBAT_INIFILE` to point to the saved hardware settings file (`.dnx`) in the `settings` subdirectory in your project directory.

For example, SET `CSPYBAT_INIFILE=C:\my_proj\settings\myproject.dnx`. Note that no quotation marks should be used around the path, even if there are blank characters.

You can also point out this file using the option `--cspybat_inifile`, like this:
`--cspybat_inifile "C:\my_proj\settings\myproject.dnx"`, see *--cspybat_inifile*, page 423.

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.xcl`, which contains options specific to `cspybat`
- `project.buildconfiguration.driver.xcl`, 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 433.
- *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 rx\bin.
<i>driver_DLL</i>	The C-SPY driver DLL file—available in rx\bin.
<i>debug_file</i>	The object file that you want to debug (filename extension out). See also <i>--debug_file</i> , page 424.
<i>cspybat_options</i>	The command line options that you want to pass to cspybat. Note that these options are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 420.
<i>--backend</i>	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 420.

Table 37: cspybat parameters

Summary of C-SPY command line options

Reference information about:

- General cspybat options
- Options available for all C-SPY drivers
- Options available for the simulator driver
- Options available for all C-SPY hardware debugger drivers
- Options available for the E1/E20 and E2/E2 Lite/EZ-CUBE2 drivers
- Options available for the J-Link driver

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>--download_only</code>	Downloads a code image without starting a debug session afterwards.
<code>-f</code>	Extends the command line.
<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>--rtc_enable</code>	Enables C-RUN runtime error checking in <code>cspybat</code> .
<code>--rtc_output</code>	Specifies to <code>cspybat</code> a file for the C-RUN message output.
<code>--rtc_raw_to_txt</code>	Makes <code>cspybat</code> act as a runtime checking message filter by reading a file as input.
<code>--rtc_rules</code>	Specifies a file for the C-RUN rules to <code>cspybat</code> .
<code>--silent</code>	Omits the sign-on message.
<code>--timeout</code>	Limits the maximum allowed execution time.

OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

<code>--core</code>	Specifies the core to be used.
<code>-d</code>	Specifies the C-SPY driver to be used.
<code>--double</code>	Specifies the size of the type <code>double</code> .
<code>--endian</code>	Specifies the byte order for data.
<code>--fpu</code>	Specifies how the code was compiled with regards to handling floating-point operations.
<code>--int</code>	Specifies the size of the type <code>int</code> .
<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>--function_profiling</code>	Analyzes your source code to find where the most time is spent during execution.
<code>--mapu</code>	Activates memory access checking.

OPTIONS AVAILABLE FOR ALL C-SPY HARDWARE DEBUGGER DRIVERS

<code>--cspybat_inifile</code>	Specifies a saved hardware settings file.
<code>--diag_warning</code>	Treats specified debugger messages as warnings.
<code>--drv_communication</code>	Identifies which emulator you are using.
<code>--drv_mode</code>	Controls the behavior of the debugger when code is downloaded.
<code>--log_file</code>	Creates a log file.
<code>--set_pc_to_entry_sym</code> <code>bol</code>	Makes the debugger start executing from the entry symbol found in the output ELF file.
<code>--suppress_download</code>	Suppresses download of the executable image.
<code>--verify_download</code>	Verifies the executable image.

OPTIONS AVAILABLE FOR THE E1/E20 AND E2/E2 LITE/EZ-CUBE2 DRIVERS

`--flash_only_changed_` Downloads just the flash memory blocks that have been
blocks changed.

OPTIONS AVAILABLE FOR THE J-LINK DRIVER

`--device_select` Selects a specific device in the JTAG scan chain.
`--ir_length` Sets the number of IR bits before the device to be debugged.

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	<code>--application_args="arg0 arg1 ..."</code>
Parameters	<div><i>arg</i> A command line argument.</div>
For use with	<code>cspybat</code>
Description	<div>Use this option to pass command line arguments to the debugged application. These variables must be defined in the application: <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></div>
Example	<code>--application_args="--logfile log.txt --verbose"</code>



To set this option, use **Project>Options>Debugger>Extra Options**

--attach_to_running_target

Syntax	<code>--attach_to_running_target</code>
For use with	<code>cspybat</code> 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.
Description	Use this option to make the debugger attach to a running application at its current location, without resetting the target system. 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.



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.

--core

Syntax	<code>--core {rxv1 rxv2 rxv3}</code>
Parameters	<code>rxv1 rxv2</code> The core you are using. This option reflects the corresponding compiler option.
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 RX</i> for information about the cores.



To set related options, choose:
Project>Options>General Options>Target>Device

--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	<code>file</code> 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 263, <i>--cycles</i> , page 423, <i>--timeout</i> , page 435.



To set this option, choose **View>Code Coverage**, right-click and choose **Save As** when the C-SPY debugger is running.

--cspybat_inifile

Syntax	<code>--cspybat_inifile <i>filename</i></code>
Parameters	<i>filename</i> The path to the hardware settings file (.dnx).
For use with	All C-SPY hardware debugger drivers.
Description	Use this option to load the saved hardware settings file (.dnx) in the <code>settings</code> subdirectory in your project directory, to use when communicating with the debugger probe.



This option is not available in the IDE.

--cycles



Syntax	<code>--cycles <i>cycles</i></code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>cycles</i> The number of cycles to run.
For use with	<code>cspybat</code>
Description	Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted. Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.



This option is not available in the IDE.

-d

Syntax	<code>-d {emue20 emue2lite jlink sim}</code>
Parameters	<code>emue20</code> Specifies the C-SPY E1/E20 driver.

	<code>emue2lite</code> Specifies the C-SPY E2 Lite or EZ-CUBE2 driver.
	<code>jlink</code> Specifies the C-SPY J-Link driver.
	<code>sim</code> Specifies the C-SPY simulator driver.
For use with	All C-SPY drivers.
Description	Use this option to specify the C-SPY driver to be used.  Project>Options>Debugger>Setup>Driver
--debug_file	
Syntax	<code>--debug_file filename</code>
Parameters	<code>filename</code> 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_select	
Syntax	<code>--device_select=position</code>
Parameters	<code>position</code> The position of the device you want to connect to.
For use with	The C-SPY J-Link driver.

Description If there is more than one device on the JTAG scan chain, use this option to select a specific device.

See also *JTAG Scan Chain*, page 446.



Project>Options>J-Link>JTAG Scan Chain>Device position

--diag_warning

Syntax `--diag_warning=tag[, tag, ...]`

Parameters

<i>tag</i>	The number of a diagnostic message found in the file <code>E20.err</code> in the installation directory, for example <code>0x0102001B</code> .
------------	--

Description Use this option to reclassify certain diagnostic messages as warnings. A warning indicates an error or omission that is of concern, but which will not cause the debugger to stop.



To set this option, use **Project>Options>Debugger>Extra Options**.

--disable_interrupts

Syntax `--disable_interrupts`

For use with The C-SPY simulator driver.

Description Use this option to disable the interrupt simulation.



To set this option, choose **Simulator>Interrupt Configuration** and deselect the **Enable interrupt simulation** command on the context menu.

--double

Syntax `--double {32|64}`

Parameters **32 (default)**
32-bit doubles are used.

	64
	64-bit doubles are used.
For use with	All C-SPY drivers.
Description	Use this option to select the precision used by the compiler for representing the floating-point types <code>double</code> and <code>long double</code> .
See also	The <i>IAR C/C++ Development Guide for RX</i> for more information about the size of the type <code>double</code> .



Project>Options>General Options>Target>Size of type ‘double’

--download_only

Syntax	<code>--download_only</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
For use with	<code>cspybat</code>
Description	Use this option to download the code image without starting a debug session afterwards.



To set a related option, choose:

Project>Options>Debugger>Setup and deselect **Run to**.


--drv_communication

Syntax	<code>--drv_communication {USB:serial_no}</code>
Parameters	<code>serial_no</code> Specifies the serial number for the USB emulator you want to use.
For use with	All C-SPY hardware debugger drivers.
Description	If you have more than one Renesas E1, E2, E2 Lite, EZ-CUBE2 or E20 emulator connected, use this option to identify which emulator you are using.
Example	<code>--drv_communication USB:9IM000019</code>



Project>Options>Driver>Communication>Serial No

--drv_mode

Syntax	<code>--drv_mode {debugging attach_to_program flash flash_and_execute}</code>
Parameters	<div>debugging</div> <p>Makes the emulator operate as a debugger. After downloading your application, you cannot disconnect the debugger and use the target system as a stand-alone unit.</p> <div>attach_to_program</div> <p>Makes the debugger continue to execute a running application at its current location, without resetting the target system. The target system must be powered by external power.</p> <div>flash</div> <p>Makes the emulator operate as a flash memory programmer. You cannot use the emulator as a debugger in this mode.</p> <div>flash_and_execute</div> <p>Launches your application on the target board when the code has been downloaded.</p>
For use with	All C-SPY hardware debugger drivers.
Description	Controls the behavior of the debugger when code is downloaded.
	 Project>Options>Driver>Download>Mode

--endian

Syntax	<code>--endian {b l}</code>
Parameters	<div>b</div> <p>Specifies big-endian as the default byte order for data.</p> <div>l (default)</div> <p>Specifies little-endian as the default byte order for data.</p>
For use with	All C-SPY drivers.
Description	Use this option to specify the byte order of the generated data. (Code is always little-endian.)

See also The *IAR C/C++ Development Guide for RX* for more information about the byte order of data.



Project>Options>General Options>Target>Byte order

-f

Syntax `-f filename`

Parameters `filename`
 A text file that contains the command line options (default filename extension `.xcl`).

For use with `cspybat`

Description Use this option to make `cspybat` 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_only_changed_blocks

Syntax `--flash_only_changed_blocks`

For use with One of these C-SPY drivers:

 ● E1/E20
 ● E2/E2 Lite/EZ-CUBE2

Description Use this option to download just the flash memory blocks that have been changed since the last download. This makes downloading faster.



Project>Options>Driver>Download>Download changed flash blocks only

--fpu

Syntax	<code>--fpu {none 32 64}</code>	
Parameters	<code>none</code>	The code was compiled to handle floating-point operations without having an FPU available.
	<code>32</code>	The code was compiled for having a 32-bit FPU available.
	<code>64</code>	The code was compiled for having a 64-bit FPU available.
For use with	All C-SPY drivers.	
Description	Use this option to specify how the code was compiled with regards to handling floating-point operations.	
See also	The <i>IAR C/C++ Development Guide for RX</i> for more information about compiling for an FPU.	



This option is set automatically when you choose:
Project>Options>General Options>Target>Device

--function_profiling

Syntax	<code>--function_profiling filename</code>	
Parameters	<code>filename</code>	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 <i>Profiling</i> , page 241.	



C-SPY driver>Function Profiling

--int

Syntax	<code>--int {16 32}</code>
Parameters	<div>16</div> <div>The size of the data type <code>int</code> is 16 bits.</div> <div>32 (default)</div> <div>The size of the data type <code>int</code> is 32 bits.</div>
For use with	All C-SPY drivers.
Description	Use this option to select whether the compiler uses 16 or 32 bits to represent the <code>int</code> data type.
See also	The <i>IAR C/C++ Development Guide for RX</i> for more information about the size of the type <code>int</code> .



Project>Options>General Options>Target>Size of type ‘int’



--ir_length

Syntax	<code>--ir_length=length</code>
Parameters	<div><i>length</i></div> <div>The the combined length in bits of all instruction registers before the device to be debugged.</div>
For use with	The C-SPY J-Link driver.
Description	Use this option to specify the combined length in bits of all instruction registers before the device to debugged, if one or more devices on the JTAG scan chain has a non-standard length (the default length is 8 bits).
See also	<i>JTAG Scan Chain</i> , page 446.




Project>Options>J-Link>JTAG Scan Chain>Preceding IR bits

--leave_target_running

Syntax	<code>--leave_target_running</code>
For use with	<p><code>cspybat</code>.</p> <p>For any of these C-SPY drivers:</p> <ul style="list-style-type: none"> ● E1/E20 ● E2/E2 Lite/EZ-CUBE2 <p>Note: Even if this option is supported by the C-SPY driver you are using, there might be device-specific limitations.</p>
Description	<p>Use this option to make the debugger leave the application running on the target hardware after the debug session is closed.</p> <div>  <p>Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this option.</p> </div> <div>  <p>C-SPY driver>Leave Target Running</p> </div>

--log_file

Syntax	<code>--log_file=filename</code>
Parameters	<p><i>filename</i></p> <p>The name of the log file.</p>
For use with	Any C-SPY hardware debugger driver.
Description	<p>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.</p> <div>  <p>Project>Options>Debugger>Driver>Communication Log</p> </div>

--macro

Syntax	<code>--macro filename</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.

Parameters	<i>filename</i> The C-SPY macro file to be used (filename extension <code>mac</code>).
For use with	<code>cspybat</code>
Description	Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.
See also	<i>Briefly about using C-SPY macros</i> , page 356.



Project>Options>Debugger>Setup>Setup macros>Use macro file

--macro_param

Syntax	<code>--macro_param [param=value]</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>param=value</i> <i>param</i> is a parameter defined using the <code>__param</code> C-SPY macro construction. <i>value</i> is a value.
For use with	<code>cspybat</code>
Description	Use this option to assign a value to a C-SPY macro parameter. This option can be used more than once on the command line.
See also	<i>Macro parameters</i> , page 363.



To set this option, use **Project>Options>Debugger>Extra Options**

--mapu

Syntax	<code>--mapu</code>
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 <code>stderr</code> and the execution will stop.

See also *Monitoring memory and registers*, page 154.



To set related options, choose:

Simulator>Memory Access Setup

-p

Syntax `-p filename`

Parameters `filename`
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 *Selecting a device description file*, page 47.



Project>Options>Debugger>Setup>Device description file

--plugin

Syntax `--plugin filename`
Note that this option must be placed before the `--backend` option on the command line.

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

For use with `cspybat`

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

Use this option to include this plugin during the debug session. This option can be used more than once on the command line.

Note: You can use this option to also include other plugin modules, but in that case the module must be able to work with `cspybat` specifically. This means that the C-SPY

plugin modules located in the `common\plugin` directory cannot normally be used with `cspybat`.



Project>Options>Debugger>Plugins

--set_pc_to_entry_symbol

Syntax	<code>--set_pc_to_entry_symbol</code>
For use with	All C-SPY hardware debugger drivers.
Description	Use this option to make the debugger start executing from the entry symbol found in the output ELF file, instead of where the device normally starts executing.



This option is not available in the IDE.

--silent

Syntax	<code>--silent</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
For use with	<code>cspybat</code>
Description	Use this option to omit the sign-on message.



This option is not available in the IDE.

--suppress_download

Syntax	<code>--suppress_download</code>
For use with	Any C-SPY hardware debugger 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>Driver>Download>Suppress download

--timeout

Syntax	<code>--timeout milliseconds</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>milliseconds</i> The number of milliseconds before the execution stops.
For use with	<code>cspybat</code>
Description	Use this option to limit the maximum allowed execution time.



This option is not available in the IDE.

--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.

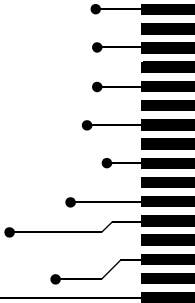


Project>Options>Debugger>Driver>Download>Verify download

Part 4. Additional reference information

This part of the *C-SPY® Debugging Guide for RX* 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 440.

- 3 On the **Setup** page, make sure to select the appropriate C-SPY driver from the **Driver** drop-down list.
- 4 To set the driver-specific options, select the appropriate driver from the **Category** list. Depending on which C-SPY driver you are using, different options are available.

C-SPY driver	Available options pages
C-SPY E1/E20 driver or	<i>Communication</i> , page 444
C-SPY E2/E2 Lite/ EZ-CUBE2 driver	<i>Download</i> , page 445
C-SPY J-Link driver	<i>Download</i> , page 445 <i>JTAG Scan Chain</i> , page 446

Table 38: 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 440
- *Images*, page 441
- *Extra Options*, page 443
- *Plugins*, page 442

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.

The screenshot shows the 'Setup' dialog box. It has a 'Driver:' dropdown menu set to 'Simulator'. To its right is a checked 'Run to:' checkbox with a text field containing 'main'. Below these are two sections: 'Setup macros' and 'Device description file'. The 'Setup macros' section has a checked 'Use macro file:' checkbox and a text field with the path '\$PROJ_DIR\$\SetupSimulation.mac'. The 'Device description file' section has a checked 'Override default:' checkbox and a text field with the path '\$TOOLKIT_DIR\$\config\debugger\generic.ddf'. Both text fields have a browse button (three dots) to their right.

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 46.

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 53.

Images

The **Images** options control the use of additional debug files to be downloaded.

The screenshot shows the 'Images' settings panel. It has a tab labeled 'Images'. Below the tab, there are three identical sections. Each section contains a checkbox labeled 'Download extra image', a text field for 'Path' with a browse button (three dots), a text field for 'Offset', and a checkbox labeled 'Debug info only'. In the first section, the 'Download extra image' checkbox is checked. In the second and third sections, it is unchecked.

Note: Images are only downloaded to RAM and no flash loading will be performed.

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

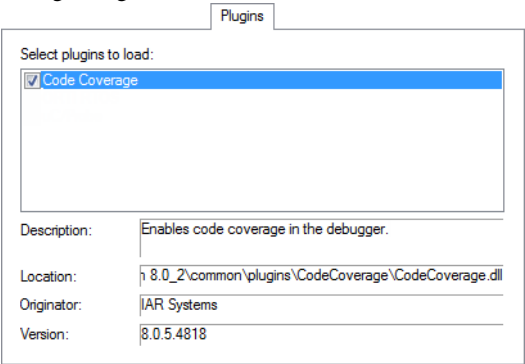
Makes the debugger download only debug information, and not the complete debug file.

If you want to download more than three debug images, use the related C-SPY macro, see `__loadImage`, page 380.

For more information, see *Loading multiple debug images*, page 49.

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 `rx\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.

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];
```

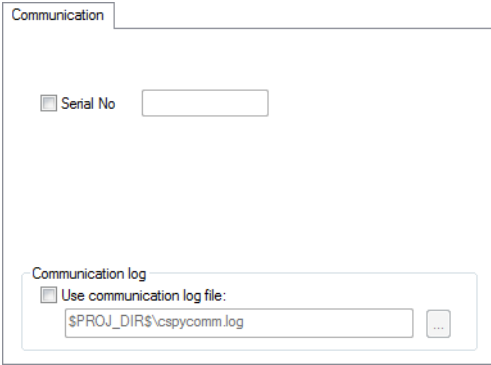
Reference information on C-SPY hardware debugger driver options

Reference information about:

- *Communication*, page 444
- *Download*, page 445
- *JTAG Scan Chain*, page 446

Communication

The **Communication** options determine how the E1, E2, E2 Lite, EZ-CUBE2 or E20 emulator communicates with the host computer.



Serial No

Selects which Renesas emulator to use, if more than one is connected to your host computer via USB.

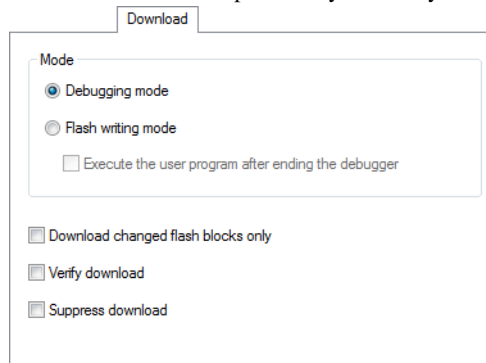
Communication log

Logs the communication between C-SPY and the target system to the specified log file, which can be useful for troubleshooting purposes. The communication will be logged in the file `cspycomm.log` located in the current working directory. If required, use the browse button to locate a different file.

To interpret the result, detailed knowledge of the interface is required.

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.



Download

Mode

☒ Debugging mode

☐ Flash writing mode

☐ Execute the user program after ending the debugger

☐ Download changed flash blocks only

☐ Verify download

☐ Suppress download

Debugging mode

Makes the emulator operate as a debugger. After downloading your application, you cannot disconnect the debugger and use the target system as a stand-alone unit. In this mode, you cannot write ID Codes (that protect the memory from being accessed) to the flash memory.

Flash writing mode

Makes the emulator operate as a flash memory programmer. You cannot use the emulator as a debugger in this mode. After downloading your application, an ID Code (that protects the memory from being accessed) is written to the flash memory.

Execute the user program after ending the debugger

Launches your application on the target board when the code has been downloaded.

Download changed flash blocks only

Downloads just the flash memory blocks that have been changed since the last download. This makes downloading faster. This option is only available for the E1/E20 and E2/E2 Lite/EZ-CUBE2 C-SPY drivers.

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 memory. 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.

JTAG Scan Chain

Only one device at a time can be debugged. If there is more than one device on the same JTAG scan chain, you must use the **J-Link>JTAG Scan Chain** options to specify which device to debug.

JTAG Scan Chain

☒ JTAG scan chain with multiple targets

Device position: 2

☒ Use devices with deviant instruction register lengths

Preceding IR bits: 12

JTAG scan chain with multiple targets

Informs the debugger that there is more than one device on the JTAG scan chain.

Device position

Specify the position of the device you want to debug. The first device has position 0.

Use devices with deviant instruction register lengths

Informs the debugger that the JTAG instruction register of one or more devices on the JTAG scan chain has a non-standard length (the default length is 8 bits).

Preceding IR bits

Specify the combined length of the instruction registers of all devices that precede the device being debugged.

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 449
- *Simulator menu*, page 450
- *E1/E20 Emulator menu*, page 452
- *E2/E2 Lite/EZ-CUBE2 menu*, page 454
- *J-Link menu*, page 457

C-SPY driver

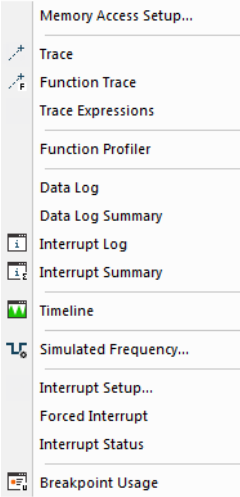
Before you start the C-SPY debugger, you must first specify a C-SPY driver in the **Options** dialog box, using the option **Debugger>Setup>Driver**.

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:

Memory Access Setup

Displays a dialog box to simulate memory access checking by specifying memory areas with different access types, see *Memory Access Setup dialog box*, page 183.



Trace

Opens a window which displays the collected trace data, see *Trace window*, page 196.



Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 203.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 249.

Data Log

Opens a window which logs accesses to up to four different memory locations or areas, see *Data Log window*, page 225.

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 228.



Interrupt Log

Opens a window which displays the status of all defined interrupts, see *Interrupt Log window*, page 345.



Interrupt Log Summary

Opens a window which displays a summary of the status of all defined interrupts, see *Interrupt Log Summary window*, page 348.



Timeline

Opens a window which gives a graphical view of various kinds of information on a timeline, see *The application timeline*, page 211.



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 459.

Interrupt Setup

Displays a dialog box where you can configure C-SPY interrupt simulation, see *Interrupt Setup dialog box*, page 339.

Forced Interrupts

Opens a window from where you can instantly trigger an interrupt, see *Forced Interrupt window*, page 342.

Interrupt Status

Opens a window from where you can instantly trigger an interrupt, see *Interrupt Status window*, page 343.

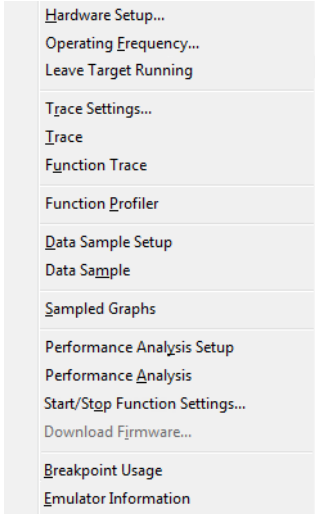


Breakpoint Usage

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 136.

EI/E20 Emulator menu

When you are using the C-SPY EI/E20 Emulator driver, the **EI/E20 Emulator** menu is added to the menu bar.



Menu commands

These commands are available on the menu:

Hardware Setup

Displays a dialog box where you can configure how the emulator operates, see *Hardware Setup dialog box: MCU*, page 63.

Operating Frequency

Displays a dialog box where you can inform the emulator of the operating frequency that the MCU is running at, see *Operating Frequency dialog box*, page 62.

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 option.

Trace Settings

Displays a dialog box where you can configure the trace generation and collection, see *Trace Settings dialog box*, page 193.

Trace

Opens a window which displays the collected trace data, see *Trace window*, page 196.

Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 203.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 249.

Data Sample Setup

Opens a window where you can specify variables to sample data for, see *Data Sample Setup window*, page 232.

Data Sample

Opens a window where you can view the result of the data sampling, see *Data Sample window*, page 230.

Sampled Graphs

Opens a window which gives a graphical view of various kinds of sampled information, see *Sampled Graphs window*, page 234.

Performance Analysis Setup

Displays a dialog box where you can configure the code performance analysis, see *Performance Analysis Setup dialog box*, page 257.

Performance Analysis

Opens a window which displays the results of the code performance analysis, see *Performance Analysis window*, page 259.

Start/Stop Function Settings

Displays a dialog box where you can configure the emulator to execute specific routines of your application immediately before the execution starts and/or after it halts, see *Start/Stop Function Settings dialog box*, page 90.

Download Firmware

Displays a dialog box where you can update the firmware of your emulator if needed, see *Download Emulator Firmware dialog box*, page 61.

Breakpoint Usage

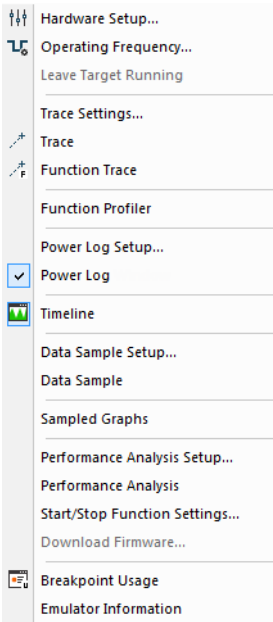
Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 136.

Emulator Information

Displays a dialog box with version information about the emulator and the emulator firmware, and related information, see *Emulator information window*, page 460.

E2/E2 Lite/EZ-CUBE2 menu

When you are using the C-SPY E2/E2 Lite/EZ-CUBE2 driver, the **E2/E2 Lite/EZ-CUBE2** menu is added to the menu bar.



Menu commands

These commands are available on the menu:

Hardware Setup

Displays a dialog box where you can configure how the emulator operates, see *Hardware Setup dialog box: MCU*, page 63.

Operating Frequency

Displays a dialog box where you can inform the emulator of the operating frequency that the MCU is running at, see *Operating Frequency dialog box*, page 62.

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 option.

Trace Settings

Displays a dialog box where you can configure the trace generation and collection, see *Trace Settings dialog box*, page 193.

Trace

Opens a window which displays the collected trace data, see *Trace window*, page 196.

Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 203.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 249.

Power Log Setup

Opens a window where you can configure the power measurement; see *Power Log Setup window*, page 278.

Power Log

Opens a window that displays collected power values; see *Power Log window*, page 281.

Timeline

Opens a window which gives a graphical view of various kinds of information on a timeline, see *Timeline window—Power graph*, page 285.

Data Sample Setup

Opens a window where you can specify variables to sample data for, see *Data Sample Setup window*, page 232.

Data Sample

Opens a window where you can view the result of the data sampling, see *Data Sample window*, page 230.

Sampled Graphs

Opens a window which gives a graphical view of various kinds of sampled information, see *Data Sample window*, page 230.

Performance Analysis Setup

Displays a dialog box where you can configure the code performance analysis, see *Performance Analysis Setup dialog box*, page 257.

Performance Analysis

Opens a window which displays the results of the code performance analysis, see *Performance Analysis window*, page 259.

Start/Stop Function Settings

Displays a dialog box where you can configure the emulator to execute specific routines of your application immediately before the execution starts and/or after it halts, see *Start/Stop Function Settings dialog box*, page 90.

Download Firmware

Displays a dialog box where you can update the firmware of your emulator if needed, see *Download Emulator Firmware dialog box*, page 61.

Breakpoint Usage

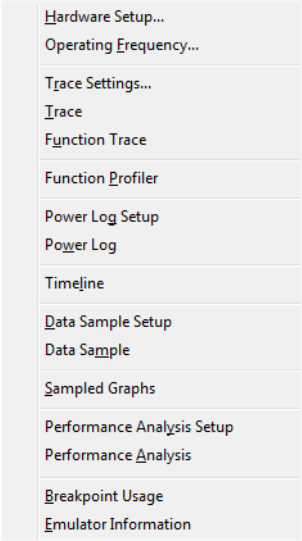
Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 136.

Emulator Information

Displays a dialog box with version information about the emulator and the emulator firmware, and related information, see *Emulator information window*, page 460.

J-Link menu

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



Menu commands

These commands are available on the menu:

Hardware Setup

Displays a dialog box where you can configure how the emulator operates, see *Hardware Setup dialog box: MCU*, page 63.

Operating Frequency

Displays a dialog box where you can inform the emulator of the operating frequency that the MCU is running at, see *Operating Frequency dialog box*, page 62.

Trace Settings

Displays a dialog box where you can configure the trace generation and collection, see *Trace Settings dialog box*, page 193.

Trace

Opens a window which displays the collected trace data, see *Trace window*, page 196.

Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 203.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 249.

Power Log Setup

Opens a window where you can configure the power measurement; see *Power Log Setup window*, page 278.

Power Log

Opens a window that displays collected power values; see *Power Log window*, page 281.

Timeline

Opens a window which gives a graphical view of various kinds of information on a timeline, see *The application timeline*, page 211.

Data Sample Setup

Opens a window where you can specify variables to sample data for, see *Data Sample Setup window*, page 232.

Data Sample

Opens a window where you can view the result of the data sampling, see *Data Sample window*, page 230.

Sampled Graphs

Opens a window which gives a graphical view of various kinds of sampled information, see *Data Sample window*, page 230.

Performance Analysis Setup

Displays a dialog box where you can configure the code performance analysis, see *Performance Analysis Setup dialog box*, page 257.

Performance Analysis

Opens a window which displays the results of the code performance analysis, see *Performance Analysis window*, page 259.

Breakpoint Usage

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 136.

Emulator Information

Displays a dialog box with version information about the emulator and the emulator firmware, and related information, see *Emulator information window*, page 460.

Reference information on the C-SPY simulator

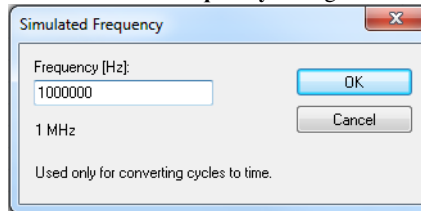
This section gives additional reference information on the C-SPY simulator, and reference information not provided elsewhere in this documentation.

Reference information about:

- *Simulated Frequency dialog box*, page 459

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.

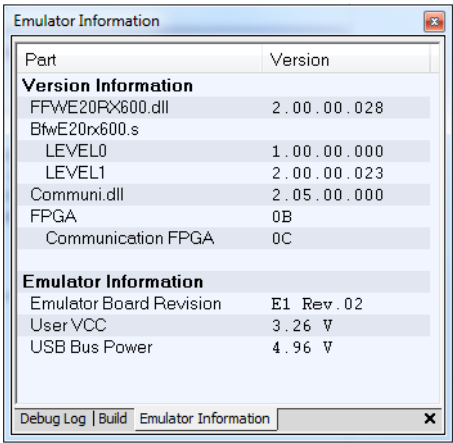
Reference information on the C-SPY hardware debugger drivers

Reference information about:

- *Emulator information window*, page 460

Emulator information window

The **Emulator information** window is available from the *C-SPY Driver* menu.



This window displays version information about the emulator and the emulator firmware, and related information.

Requirements

A C-SPY hardware debugger driver.

Display area

This area displays version information and other information.

Version Information

Displays information about the emulator firmware DLL, the communication DLL, and the FPGA circuit.

Emulator Information

Displays the version and revision number of the emulator hardware, the actual voltage that the target board is powered with, and the voltage of the USB bus.

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)	89
__abortLaunch (C-SPY system macro)	372
absolute location, specifying for a breakpoint	149
Access type (Edit Memory Access option)	186
Access (Edit SFR option)	182
accesses outside the bounds of arrays and other objects, detecting	301
Action (Hardware Code breakpoints option)	139
Action (Software Code breakpoints option)	140
Address Range (Find in Trace option)	208
Address range (Hardware Setup)	68
Address (Edit SFR option)	182
Allow clock source change when writing internal flash (Hardware Setup)	65
Ambiguous symbol (Resolve Symbol Ambiguity option)	122
--application_args (C-SPY command line option)	420
application, built outside the IDE	48
assembler labels, viewing	102
assembler source code, fine-tuning	241
assembler symbols, in C-SPY expressions	99
assembler variables, viewing	102
assumptions, programming experience	23
Attach to program (debugger option) example	52
Auto Scroll (Timeline window context menu)	220, 223, 287, 351
Auto window	103
Autostep settings dialog box	93

B

--attach_to_running_target (C-SPY command line option)	421
--backend (C-SPY command line option)	421
backtrace information, viewing in Call Stack window	84
batch mode, using C-SPY in	415
Big Endian (Memory window context menu)	162

bit loss or undefined behavior when shifting, detecting	299
blocks, in C-SPY macros	365
bold style, in this guide	28
bounds of arrays and other objects, detect accesses outside	301
--bounds_table_size (linker option)	321
Break At (Hardware Code breakpoints option)	139
Break At (Software Code breakpoints option)	140
breakpoint condition, example	132–133
breakpoint dialog box Code	137
Data	142
Data Log	146
Immediate	147
Log	141
Trace Start Trigger	205
Trace Stop Trigger	206
Breakpoint Usage window	136
Breakpoint Usage (J-Link Emulator menu)	458
breakpoints briefly about	123
code, example	392
connecting a C-SPY macro	360
consumers of	127
data	142
data log	146
description of	123
disabling used by Stack window	128
icons for in the IDE	126
in Memory window	131
listing all	136
reasons for using	123
setting in memory window	131
using system macros	131
using the dialog box	129
single-stepping if not available	46
toggling	129
types of	124
useful tips	132

Breakpoints dialog box	
Data Trace Collection	207
Data (C-SPY hardware debugger drivers)	144
Hardware Code	138
Performance Start	261
Performance Stop	262
Software Code	140
Breakpoints window	134
Browse (Trace toolbar)	197
Bus width (External Area option)	68
Byte order (External Area option)	68
Byte order (Hardware Setup)	64
byte order, setting in Memory window	161

C

C function information, in C-SPY	77
C symbols, in C-SPY expressions	99
C variables, in C-SPY expressions	98
call chain, displaying in C-SPY	77
Call Stack graph (Timeline window)	218
Call stack information	77
Call Stack window	84
for backtrace information	77
Call Stack (Timeline window context menu)	220
__cancelAllInterrupts (C-SPY system macro)	372
__cancelInterrupt (C-SPY system macro)	372
checked heap, using	291
Clear Group	
(Registers User Groups Setup window context menu)	177
Clear trace data (Trace toolbar)	196
__clearBreak (C-SPY system macro)	373
clock frequency, simulated	459
__closeFile (C-SPY system macro)	373
code	
options for downloading	445
code breakpoints	
overview	124
toggling	129

code coverage	
using	264
Code Coverage window	265
--code_coverage_file (C-SPY command line option)	422
code, covering execution of	265
Collected data accesses (Trace Settings option)	195
command line options	420
typographic convention	28
command prompt icon, in this guide	28
Communication log (debugger option)	445
communication setup, hardware drivers	444
Communication (Hardware Setup)	65
computer style (monospace font), typographic convention	27
Condition (Performance Analysis option)	257
conditional statements, in C-SPY macros	364
context menu, in windows	101
conventions, used in this guide	27
Copy Window Contents	
(Disassembly window context menu)	83
copyright notice	2
--core (C-SPY command line option)	422
Core (Cores window)	94
cores	
inspecting state of	94
Cores window	94
cspybat	415
reading options from file (-f)	428
--cspybat_inifile (C-SPY command line option)	423
CSPYBAT_INIFILE (environment variable)	415
current position, in C-SPY Disassembly window	80
cursor, in C-SPY Disassembly window	80
Cycle costs (Edit Memory Access option)	186
--cycles (C-SPY command line option)	423
Cycles (Cores window)	95
C-RUN	
creating rules for messages	295
detecting various runtime errors	295
getting started	293
in non-interactive mode	292
in the IDE	291

- requirements for 292
- setting options for 314
- using 292
- using the checked heap 291
- C-RUN Messages Rules window 318
- C-RUN Messages window 316
- C-RUN runtime error checking 289
- C-RUN runtime error checking, documentation 26
- C-SPY
 - batch mode, using in 415
 - debugger systems, overview of 37
 - differences between drivers 39
 - environment overview 33
 - plugin modules, loading 47
 - scripting. *See* macros
 - setting up 45–46
 - starting the debugger 47
- C-SPY drivers
 - E1/E20 41
 - E2/E2 Lite 41
 - J-Link 42
 - overview 39
 - specifying 440
 - types of 38
- C-SPY expressions 98
 - evaluating, using Macro Quicklaunch window 412
 - evaluating, using Quick Watch window 116
 - in C-SPY macros 364
 - Tooltip watch, using 97
 - Watch window, using 97
- C-SPY macros
 - blocks 365
 - conditional statements 364
 - C-SPY expressions 364
 - examples 357
 - checking status of register 359
 - creating a log macro 360
 - executing 357
 - connecting to a breakpoint 360
 - using Quick Watch 359
 - using setup macro and setup file 359
 - functions 100, 362
 - keywords 362–363, 365
 - loop statements 365
 - macro statements 364
 - parameters 363
 - setup macro file 356
 - executing 359
 - setup macro functions 356
 - summary 367
 - system macros, summary of 369
 - using 355
 - variables 100, 362
- C-SPY options
 - Extra Options 443
 - Images 441
 - Plugins 442
 - Setup 440
- C-SPYLink 39
- C-STAT for static analysis, documentation for 26
- C++ exceptions
 - debugging 58
 - single stepping 72

D

- d (C-SPY command line option) 423
- data breakpoints, overview 125
- Data Coverage (Memory window context menu) 162
- data coverage, in Memory window 160
- data log breakpoints, overview 125
- Data Log graph (Timeline window) 221
- Data Log Summary window 228
- Data Log window 225
- Data Log (Timeline window context menu) 223
- Data Sample Setup dialog box (E1/E20 Emulator menu) 453
- Data Sample Setup dialog box (E2/E2 Lite menu) 455
- Data Sample Setup dialog box (J-Link Emulator menu) 458

Data Sample Setup window	232	description (interrupt property)	341
Data Sample Setup dialog box (J-Link Emulator menu)	458	Device description file (debugger option)	441
Data Sample window	230	device description files	47
Data Sample window (J-Link Emulator menu)	458	definition of	53
Data Sample (Sampled Graphs window context menu) ..	236	memory zones	153
Data to collect (Trace Settings option)	195	modifying	53
data trace collection breakpoints, overview	125	register zone	153
ddf (filename extension), selecting a file	47	specifying interrupts	386
Debug Log window	88	Device position (J-Link debugger option)	446
Debug menu (C-SPY main window)	56	--device_select (C-SPY command line option)	424
Debug the program re-writing the DATA FLASH (Hardware Setup)	66	diagnostic messages	
Debug the program re-writing the PROGRAM ROM (Hardware Setup)	65	classifying as C-SPY warnings	425
Debug (Report Assert option)	89	--diag_warning (C-SPY command line option)	425
--debug_file (cspybat option)	424	__disableInterrupts (C-SPY system macro)	374
debugger concepts, definitions of	36	disable_check (pragma directive)	324
debugger drivers		--disable_interrupts (C-SPY command line option)	425
differences between	39	Disassembly window	79
E1/E20	41	context menu	81
E2/E2 Lite	41	disclaimer	2
J-Link	42	Display the cycle as a time span (Performance Analysis option)	259
simulator	40	Display timestamp (Trace Settings option)	195
Debugger Macros window	410	division by zero, detecting	300
debugger system overview	37	DLIB	
Debugging mode (debugger option)	445	consuming breakpoints	128
debugging projects		naming convention	29
externally built applications	48	do (macro statement)	365
loading multiple images	49	document conventions	27
debugging, RTOS awareness	35	documentation	
--debug_heap (linker option)	321	overview of guides	25
default_no_bounds (pragma directive)	323	overview of this guide	24
define_without_bounds (pragma directive)	324	this guide	23
define_with_bounds (pragma directive)	324	--double (C-SPY command line option)	425
__delay (C-SPY system macro)	374	Download changed flash blocks only (debugger option) ..	446
Delay (Autostep Settings option)	93	Download Firmware (E1/E20 Emulator menu)	453
Delete/revert All Custom SFRs (SFR Setup window context menu)	180	Download Firmware (E2/E2 Lite menu)	456
Description (Edit Interrupt option)	341	--download_only (C-SPY command line option)	426
		Driver (debugger option)	440
		__driverType (C-SPY system macro)	374
		--drv_communication (C-SPY command line option)	426

--drv_mode (C-SPY command line option) 427

E

Edit Breakpoint. 83
 Edit Interrupt dialog box. 341
 Edit Memory Access dialog box. 185
 Edit Memory Range dialog box 181
 Edit Settings (Trace toolbar). 197
 edition, of this guide 2
 emulator firmware, updating. 61
 Emulator mode (Hardware Setup). 66
 emulator, getting information about 460
 Enable interrupt simulation
 (Interrupt Setup option). 339
 Enable runtime checking (C-RUN option) 314, 317
 Enable start routine (Start/Stop Function Settings option) . 91
 Enable stop routine (Start/Stop Function Settings option). . 92
 Enable (Sampled Graphs window context menu) 236
 __enableInterrupts (C-SPY system macro). 375
 Enable/Disable Breakpoint
 (Disassembly window context menu). 83
 Enable/Disable (Trace toolbar) 196
 End address (Memory Save option) 163
 --endian (C-SPY command line option) 427
 endianness. *See* byte order
 Enter Location dialog box. 148
 Erase data flash ROM before download (Hardware Setup). 65
 Erase external flash
 ROM before download (Hardware Setup) 68
 Erase flash ROM before download (Hardware Setup) . . . 65
 error checking (C-RUN), documentation 26
 __evaluate (C-SPY system macro) 375
 Evaluate Now
 (Macro Quicklaunch window context menu) 413
 examples
 C-SPY macros 357
 interrupts
 interrupt logging 338
 timer 336

macros
 checking status of register. 359
 creating a log macro 360
 using Quick Watch 359
 performing tasks and continue execution 133
 tracing incorrect function arguments 132
 execUserAttach (C-SPY setup macro) 367
 execUserExit (C-SPY setup macro) 369
 execUserPreload (C-SPY setup macro) 368
 execUserPreReset (C-SPY setup macro). 368
 execUserReset (C-SPY setup macro) 369
 execUserSetup (C-SPY setup macro) 368
 Execute the user program
 after ending the debugger (debugger option) 445
 executed code, covering 265
 execution history, tracing 192
 Execution state (Cores window) 94
 expressions. *See* C-SPY expressions
 EXTAL frequency (Hardware Setup) 64
 extended command line file, for cspybat. 428
 External Flash Definition Editor (Renesas tool) 51
 External flash definition file (Hardware Setup). 67
 external flash definition files. 51
 external flash memory
 downloading to 51
 External memory areas (Hardware Setup) 64
 external memory area, defining. 68
 Extra Options, for C-SPY 443
 E1/E20 Emulator (C-SPY driver), menu. 452
 E1/E20 (C-SPY driver). 41
 hardware installation 42
 E2 Lite (C-SPY driver)
 menu 454
 E2/E2 Lite (C-SPY driver) 41

F

-f (cspybat option). 428
 File format (Memory Save option) 163

file types	
device description, specifying in IDE	47
macro	46, 441
filename extensions	
ddf, selecting device description file	47
mac, using macro file	46
Filename (Memory Restore option)	164
Filename (Memory Save option)	164
Fill dialog box	165
__fillMemory8 (C-SPY system macro)	376
__fillMemory16 (C-SPY system macro)	377
__fillMemory32 (C-SPY system macro)	378
Find in Trace dialog box	207
Find in Trace window	209
Find in Trace (Disassembly window context menu)	84
Find (Memory window context menu)	162
Find (Trace toolbar)	197
firmware (emulator), updating	61
first activation time (interrupt property), definition of	332
First activation (Edit Interrupt option)	341
flash memory, load library module to	381
Flash writing mode (debugger option)	445
--flash_only_changed_blocks	
(C-SPY command line option)	428
__fmessage (C-SPY macro keyword)	365
for (macro statement)	365
Force (Forced Interrupt window context menu)	343
Forced Interrupt window	342
Forced Interrupts (Simulator menu)	451
Format	
(Registers User Groups Setup window context menu)	177
--fpu (compiler option)	429
Frequency ratio (Trace Settings option)	195
Function Profiler window	249
Function Profiler (E1/E20 Emulator menu)	453
Function Profiler (E2/E2 Lite menu)	455
Function Profiler (J-Link Emulator menu)	458
Function Profiler (Simulator menu)	450
Function Trace window	203
Function Trace (E1/E20 Emulator menu)	453

Function Trace (E2/E2 Lite menu)	455
Function Trace (J-Link Emulator menu)	458
functions	
C-SPY running to when starting	46, 440
most time spent in, locating	241
--function_profiling (cspybat option)	429

G

--generate_entries_without_bounds (compiler option)	321
generate_entry__without_bounds (pragma directive)	325
__getNumberOfCores (C-SPY system macro)	379
__getSelectedCore (C-SPY system macro)	379
Go To Source	
(Timeline window context menu)	220, 224, 288, 352
Go (Debug menu)	75

H

hardware code breakpoints, overview	125
Hardware Setup dialog box	63, 66
Hardware Setup (E1/E20 Emulator menu)	452
Hardware Setup (E2/E2 Lite menu)	454
Hardware Setup (J-Link Emulator menu)	457
hardware setup, power consumption because of	274
heap	291
heap integrity violations, detecting	311
heap memory leaks, detecting	309
heap usage error, detecting	308
highlighting, in C-SPY	76
High-performance Embedded Workshop, migrating from	26
Hold time (Edit Interrupt option)	342
hold time (interrupt property), definition of	333

I

ICLK frequency (Hardware Setup)	64
icons, in this guide	28
ID Code verification	93

- if else (macro statement) 364
- if (macro statement) 364
- Ignore (Report Assert option) 90
- ignore_uninstrumented_pointers (compiler option) 322
- ignore_uninstrumented_pointers (linker option) 322
- Images window 59
- Images, loading multiple 441
- immediate breakpoints, overview 126
- implicit or explicit integer conversion, detecting 295
- Input Mode dialog box 87
- input, special characters in Terminal I/O window 87
- insert checks for (C-RUN option) 316
- installation directory 27
- Instruction Profiling (Disassembly window context menu) 82
- int (C-SPY command line option) 430
- integer conversion, detect implicit or explicit 295
- Intel-extended, C-SPY output format 38
- interference, power consumption because of 275
- interrupt handling, power consumption during 273
- Interrupt Log graph in Timeline window 350
- Interrupt Log Summary window 348
- Interrupt Log window 345
- Interrupt Setup dialog box 339
- Interrupt Setup (Simulator menu) 451
- Interrupt Status window 343
- interrupt system, using device description file 335
- Interrupt (Edit Interrupt option) 341
- interrupts
 - adapting C-SPY system for target hardware 335
 - simulated, introduction to 331
 - timer, example 336
 - using system macros 334
- Interrupts (Timeline window context menu) 352
- ir_length (C-SPY command line option) 430
- __isBatchMode (C-SPY system macro) 379
- __isMacroSymbolDefined (C-SPY system macro) 380
- italic style, in this guide 27–28
- I/O register. *See* SFR

J

- JTAG scan chain with multiple targets (J-Link debugger option) 446
- J-Link (C-SPY driver) 42
 - hardware installation 43
 - menu 457

L

- labels (assembler), viewing 102
- Leave Target Running (C-SPY driver menu command) 452
- Leave Target Running (E2/E2 Lite menu) 455
- leave_target_running (C-SPY command line option) 431
- Length (Fill option) 165
- library functions
 - C-SPY support for using, plugin module 433
- lightbulb icon, in this guide 28
- linker options
 - typographic convention 27
 - consuming breakpoints 128
- Little Endian (Memory window context menu) 161
- Live Watch window 111
- __loadImage (C-SPY system macro) 380
- loading multiple debug files, list currently loaded 59
- loading multiple images 49
- Locals window 106
- log breakpoints, overview 124
- log_file (C-SPY command line option) 431
- loop statements, in C-SPY macros 365
- low-power mode, power consumption during 272

M

- mac (filename extension), using a macro file 46
- macro (C-SPY command line option) 431
- macro files, specifying 46, 441
- Macro Quicklaunch window 412
- Macro Registration window 408

macro statements	364
macros	
executing	357
using	355
--macro-param (C-SPY command line option)	432
main function, C-SPY running to when starting	46, 440
--mapu (C-SPY command line option)	432
MCU operation, configuring	63–64
MCU speed, specifying	62
Measure the performance only	
once (Performance Analysis option)	259
Memory access checking (Memory Access Setup option)	184
Memory Access Setup dialog box	183
Memory Access Setup (Simulator menu)	450
Memory Fill (Memory window context menu)	162
memory map	183
Memory Restore dialog box	164
Memory Restore (Memory window context menu)	162
Memory Save dialog box	163
Memory Save (Memory window context menu)	162
Memory window	159
memory zones	153
in device description file	153
__memoryRestore (C-SPY system macro)	382
__memorySave (C-SPY system macro)	382
menu bar, C-SPY-specific	56
__message (C-SPY macro keyword)	365
__messageBoxYesCancel (C-SPY system macro)	383
__messageBoxYesNo (C-SPY system macro)	384
Messages window, amount of output	88
migration	
from a UBROF-based product	26
from Renesas HEW	26
migration, from earlier IAR compilers	26
Mixed Mode (Disassembly window context menu)	83
monospace font, meaning of in guide. <i>See</i> computer style	
Motorola, C-SPY output format	38
Move to PC (Disassembly window context menu)	81

N

Name (Edit SFR option)	181
naming conventions	28
Navigate	
(Sampled Graphs window context menu)	235
Navigate	
(Timeline window context menu)	219, 223, 286, 351
Next Symbol (Symbolic Memory window context menu)	168
no_arith_checks (pragma directive)	325
no_bounds (pragma directive)	325

O

Open Setup Window	
(Timeline window context menu)	288
__openFile (C-SPY system macro)	385
Operating Frequency (E1/E20 Emulator menu)	452
Operating Frequency (E2/E2 Lite menu)	454
Operating Frequency (J-Link Emulator menu)	457
operating frequency, specifying	62
Operation (Fill option)	165
operators, sizeof in C-SPY	100
optimizations, effects on variables	100
options	
in the IDE	439
on the command line	420, 443
Options (Stack window context menu)	172
__orderInterrupt (C-SPY system macro)	386
Originator (debugger option)	442
overflow, signed or unsigned	297

P

-p (C-SPY command line option)	433
__param (C-SPY macro keyword)	363
parameters	
tracing incorrect values of	77
typographic convention	27

part number, of this guide 2

PC (Cores window). 94

Performance Analysis Setup dialog box 257

Performance Analysis Setup (E1/E20 menu) 453

Performance Analysis Setup (E2/E2 Lite menu). 456

Performance Analysis Setup (J-Link Emulator menu) . . . 458

Performance Analysis window 259

Performance Analysis (E1/E20 Emulator menu) 453

Performance Analysis (E2/E2 Lite menu). 456

Performance Analysis (J-Link Emulator menu) 458

performance start and stop breakpoints, overview 126

Performance Start breakpoints dialog box 261

Performance Stop breakpoints dialog box. 262

peripheral units

- debugging power consumption for. 269
- detecting mistakenly unattended 273
- detecting unattended 273
- device-specific 54
- displayed in Registers window. 152
- in an event-driven system 273
- in C-SPY expressions 99
- initializing using setup macros. 356

peripherals register. *See* SFR

Please select one symbol

- (Resolve Symbol Ambiguity option) 122

--plugin (C-SPY command line option) 433

plugin modules (C-SPY). 38

- loading. 47

Plugins (C-SPY options). 442

__popSimulatorInterruptExecutingStack

- (C-SPY system macro) 387

pop-up menu. *See* context menu

power consumption, measuring 242, 269

Power Log Setup window. 278

Power Log Setup (J-Link Emulator menu) 455, 458

Power Log window. 281

Power Log (J-Link Emulator menu) 455, 458

Power Log (Timeline window context menu). 287

power sampling. 242

Power target from the emulator (Hardware Setup) 66

Preceding IR bits (J-Link debugger option) 447

prerequisites, programming experience 23

Previous Symbol

- (Symbolic Memory window context menu) 168

probability (interrupt property) 342

- definition of. 332

Probability % (Edit Interrupt option) 342

Profile Selection (Timeline window context menu) . 221, 288

profiling

- analyzing data 244
- on function level 243
- on instruction level. 246

profiling information, on functions and instructions 241

profiling sources

- sampling 242
- trace (calls) 242
- trace (flat) 242

program execution

- breaking. 124–125
- in C-SPY 71

programming experience 23

program. *See* application

Progress bar (Trace toolbar) 197

projects, for debugging externally built applications. 48

publication date, of this guide. 2

Q

Quick Watch window 116

- executing C-SPY macros 359

R

Range for (Viewing Range option) 238

__readFile (C-SPY system macro) 387

__readFileByte (C-SPY system macro) 388

reading guidelines. 23

__readMemoryByte (C-SPY system macro). 388

__readMemory8 (C-SPY system macro) 388

<code>__readMemory16</code> (C-SPY system macro)	389
<code>__readMemory32</code> (C-SPY system macro)	389
reference information, typographic convention.	28
register groups	152
predefined, enabling.	173
Register setting (Hardware Setup)	64
Register User Groups Setup window	176
registered trademarks	2
<code>__registerMacroFile</code> (C-SPY system macro)	390
Registers window	173
registers, displayed in Registers window	173
Removal All Groups (Registers User Groups Setup window context menu) . . .	177
Removal (Registers User Groups Setup window context menu) . . .	177
Remove All (Macro Quicklaunch window context menu) . . .	413
Remove (Macro Quicklaunch window context menu) . . .	413
Renesas HEW, migrating from	26
Repeat interval (Edit Interrupt option)	341
repeat interval (interrupt property), definition of	332
Replace (Memory window context menu)	162
Report Assert dialog box	89
<code>__resetFile</code> (C-SPY system macro).	390
Resolve Source Ambiguity dialog box	150
Restart emulator when trace buffer is full (Trace Settings option).	194
Restore (Memory Restore option).	164
return (macro statement).	365
ROM-monitor, definition of	38
RTOS awareness debugging	35
RTOS awareness (C-SPY plugin module)	35
Run to Cursor (Disassembly window context menu)	81
Run to Cursor, command for executing	76
Run to (C-SPY option)	46, 440
runtime checking, setting options for C-RUN.	314
runtime error checking	289
getting started	
requirements for	293
requirements for.	292
using C-RUN.	290

runtime error checking, documentation.	26
--runtime_checking (compiler option)	322

S

Sampled Graphs window	234
sampling, profiling source	242
Save Custom SFRs (SFR Setup window context menu) . . .	181
Save to File (Timeline window context menu)	220
Save to File (Register User Groups Setup window context menu)	178
Save (Memory Save option)	164
Save (Trace toolbar)	197
Scale (Viewing Range option)	239
scripting C-SPY. <i>See</i> macros	
Select Graphs (Sampled Graphs window context menu). . .	237
Select Graphs (Timeline window context menu)	220, 224, 288, 352
Select plugins to load (debugger option).	442
<code>__selectCore</code> (C-SPY system macro)	391
Serial No (E1/E20 option)	444
Set Data Breakpoint (Memory window context menu). . . .	162
Set Data Log Breakpoint (Memory window context menu)	163
Set Next Statement (Disassembly window context menu) . .	83
<code>__setCodeBreak</code> (C-SPY system macro).	391
<code>__setDataBreak</code> (C-SPY system macro)	392
<code>__setDataLogBreak</code> (C-SPY system macro)	393
<code>__setLogBreak</code> (C-SPY system macro)	394
<code>__setSimBreak</code> (C-SPY system macro)	395
<code>__setTraceStartBreak</code> (C-SPY system macro)	396
<code>__setTraceStopBreak</code> (C-SPY system macro).	397
setup macro file, registering	46
setup macro functions	356
reserved names.	367
Setup macros (debugger option)	441
Setup (C-SPY options)	440
--set_pc_to_entry_symbol (C-SPY command line option) . .	434
SFR in Registers window.	174

- using as assembler symbols 99
- SFR Setup window 178
- shifting, detecting bit loss or undefined behavior 299
- shortcut menu. *See* context menu
- Show All (SFR Setup window context menu). 180
- Show Custom SFRs only
(SFR Setup window context menu) 180
- Show Factory SFRs only
(SFR Setup window context menu) 180
- Show Numerical Value
(Timeline window context menu). 224, 237, 288
- Show offsets (Stack window context menu). 171
- Show Timing (Timeline window context menu). 220
- Show variables (Stack window context menu). 171
- signed or unsigned overflow, detecting. 297
- silent (C-SPY command line option) 434
- Simulated Frequency dialog box. 459
- simulating interrupts, enabling/disabling 339
- Simulator menu. 450
- simulator, introduction 40
- Size (Edit SFR option) 182
- Size (Sampled Graphs window context menu) 237
- Size (Timeline window context menu). 224, 287
- sizeof 100
- __smessage (C-SPY macro keyword). 365
- software code breakpoints, overview 125
- software delay, power consumption during. 271
- Solid Graph (Sampled Graphs window context menu) 237
- Solid Graph (Timeline window context menu). 224, 288
- Sort by (Timeline window context menu). 352
- __sourcePosition (C-SPY system macro) 398
- special function registers (SFR)
 - in Registers window. 174
 - using as assembler symbols 99
- Stack window 169
- standard C, sizeof operator in C-SPY 100
- Start address (Fill option) 165
- Start address (Memory Save option). 163
- Start routine location (Start/Stop Function Settings option) 91
- Start/Stop Function Settings dialog box 90

- static analysis tool, documentation for 26
- Statics window 113
- Status (Cores window) 94
- Step Into, description 73
- Step Out, description. 74
- Step Over, description. 73
- step points, definition of 72
- Stop routine location (Start/Stop Function Settings option) 92
- __strFind (C-SPY system macro) 398
- __subString (C-SPY system macro) 399
- Suppress download (debugger option) 446
- suppress_download (C-SPY command line option) 434
- switch, detect unhandled cases 300
- Symbolic Memory window. 166
- Symbols window 119
- symbols, in C-SPY expressions 98
- __system1 (C-SPY system macro) 400
- __system2 (C-SPY system macro) 400
- __system3 (C-SPY system macro) 401

T

- target board, actual voltage 460
- target system, definition of 37
- __targetDebuggerVersion (C-SPY system macro) 402
- Terminal IO Log Files (Terminal IO Log Files option) . . . 87
- Terminal I/O Log Files dialog box 87
- Terminal I/O window 77, 86
- Text search (Find in Trace option) 208
- Time Axis Unit
(Timeline window context menu). 220, 224, 288, 352
- time interval, in Timeline window 247
- Timeline window 350
- Timeline window (Call Stack graph) 218
- Timeline window (Data Log graph) 221
- Timeline (E2/E2 Lite menu). 455
- Timeline (J-Link Emulator menu). 458
- timeout (C-SPY command line option) 435
- timer interrupt, example 336

timestamp of collected data	
displaying	199
option to set	195
Toggle Breakpoint (Code)	
(Disassembly window context menu)	82
Toggle Breakpoint (Hardware (Code)	
(Disassembly window context menu)	82
Toggle Breakpoint (Log)	
(Disassembly window context menu)	83
Toggle Breakpoint (Performance Start)	
(Disassembly window context menu)	82
Toggle Breakpoint (Performance Stop)	
(Disassembly window context menu)	83
Toggle Breakpoint (Software Code)	
(Disassembly window context menu)	82
Toggle Breakpoint (Trace Start)	
(Disassembly window context menu)	83
Toggle Breakpoint (Trace Stop)	
(Disassembly window context menu)	83
Toggle source (Trace toolbar)	196
__toLower (C-SPY system macro)	402
tools icon, in this guide	28
__toString (C-SPY system macro)	403
__toUpper (C-SPY system macro)	403
trace	189, 211
Trace capacity (Trace Settings option)	194
Trace mode (Trace Settings option)	193
Trace output (Trace Settings option)	194
Trace Settings dialog box	193
Trace Settings (E1/E20 Emulator menu)	452
Trace Settings (E2/E2 Lite menu)	455
Trace Settings (J-Link Emulator menu)	457
Trace Start Trigger breakpoint dialog box	205
trace start/stop trigger breakpoints, overview	124
Trace Stop Trigger breakpoint dialog box	206
Trace type (Trace Settings option)	194
Trace window	196
trace (calls), profiling source	242
Trace (emulator mode)	66
Trace (E1/E20 Emulator menu)	453

Trace (E2/E2 Lite menu)	455
trace (flat), profiling source	242
Trace (J-Link menu)	457
trademarks	2
typographic conventions	27

U

Unavailable, C-SPY message	101
unhandled cases in switch statements, detecting	300
__unloadImage (C-SPY system macro)	404
USB bus, actual voltage	460
USD files	51
Use command line options (debugger option)	443
Use devices with deviant instruction register lengths (J-Link debugger option)	447
Use Extra Images (debugger option)	441
Use manual ranges (Memory Access Setup option)	184
Use ranges based on (Memory Access Setup option)	183
Use 64-bit counter (Performance Analysis option)	259
user application, definition of	37
using checked variant	291

V

Value (Fill option)	165
__var (C-SPY macro keyword)	362
variables	
effects of optimizations	100
in C-SPY expressions	98
information, limitation on	100
variance (interrupt property), definition of	333
Variance % (Edit Interrupt option)	342
Verify download (debugger option)	446
--verify_download (C-SPY command line option)	435
version number	
of this guide	2
Viewing Range dialog box	238
Viewing Range (Sampled Graphs window context menu)	236

Viewing Range
(Timeline window context menu) 224, 287
Visual State, C-SPY plugin module for 39

W

waiting for device, power consumption during 271
__wallTime_ms (C-SPY system macro) 404
warnings
 classifying in C-SPY 425
warnings icon, in this guide 28
Watch window 108
 using 97
web sites, recommended 27
while (macro statement) 365
windows, specific to C-SPY 58
Work RAM start address (Hardware Setup) 65
__writeFile (C-SPY system macro) 405
__writeFileByte (C-SPY system macro) 405
__writeMemoryByte (C-SPY system macro) 406
__writeMemory8 (C-SPY system macro) 406
__writeMemory16 (C-SPY system macro) 406
__writeMemory32 (C-SPY system macro) 407

Z

zone
 defined in device description file 153
 in C-SPY 153
 part of an absolute address 149
Zone (Edit SFR option) 182
Zoom (Sampled Graphs window context menu) 236
Zoom
(Timeline window context menu) 220, 223, 287, 352

Symbols

__abortLaunch (C-SPY system macro) 372
__as_get_base (operator) 326

__as_get_bound (operator) 326
__as_make_bounds (operator) 326
__cancelAllInterrupts (C-SPY system macro) 372
__cancelInterrupt (C-SPY system macro) 372
__clearBreak (C-SPY system macro) 373
__closeFile (C-SPY system macro) 373
__delay (C-SPY system macro) 374
__disableInterrupts (C-SPY system macro) 374
__driverType (C-SPY system macro) 374
__enableInterrupts (C-SPY system macro) 375
__evaluate (C-SPY system macro) 375
__fillMemory8 (C-SPY system macro) 376
__fillMemory16 (C-SPY system macro) 377
__fillMemory32 (C-SPY system macro) 378
__fmessage (C-SPY macro keyword) 365
__getNumberOfCores (C-SPY system macro) 379
__getSelectedCore (C-SPY system macro) 379
__isBatchMode (C-SPY system macro) 379
__isMacroSymbolDefined (C-SPY system macro) 380
__loadImage (C-SPY system macro) 380
__memoryRestore (C-SPY system macro) 382
__memorySave (C-SPY system macro) 382
__message (C-SPY macro keyword) 365
__messageBoxYesCancel (C-SPY system macro) 383
__messageBoxYesNo (C-SPY system macro) 384
__openFile (C-SPY system macro) 385
__orderInterrupt (C-SPY system macro) 386
__param (C-SPY macro keyword) 363
__popSimulatorInterruptExecutingStack
(C-SPY system macro) 387
__readFile (C-SPY system macro) 387
__readFileByte (C-SPY system macro) 388
__readMemoryByte (C-SPY system macro) 388
__readMemory8 (C-SPY system macro) 388
__readMemory16 (C-SPY system macro) 389
__readMemory32 (C-SPY system macro) 389
__registerMacroFile (C-SPY system macro) 390
__resetFile (C-SPY system macro) 390
__selectCore (C-SPY system macro) 391
__setCodeBreak (C-SPY system macro) 391

__setDataBreak (C-SPY system macro)	392
__setDataLogBreak (C-SPY system macro)	393
__setLogBreak (C-SPY system macro)	394
__setSimBreak (C-SPY system macro)	395
__setTraceStartBreak (C-SPY system macro)	396
__setTraceStopBreak (C-SPY system macro)	397
__smessage (C-SPY macro keyword)	365
__sourcePosition (C-SPY system macro)	398
__strFind (C-SPY system macro)	398
__subString (C-SPY system macro)	399
__system1 (C-SPY system macro)	400
__system2 (C-SPY system macro)	400
__system3 (C-SPY system macro)	401
__targetDebuggerVersion (C-SPY system macro)	402
__toLower (C-SPY system macro)	402
__toString (C-SPY system macro)	403
__toUpper (C-SPY system macro)	403
__unloadImage (C-SPY system macro)	404
__var (C-SPY macro keyword)	362
__wallTime_ms (C-SPY system macro)	404
__writeFile (C-SPY system macro)	405
__writeFileByte (C-SPY system macro)	405
__writeMemoryByte (C-SPY system macro)	406
__writeMemory8 (C-SPY system macro)	406
__writeMemory16 (C-SPY system macro)	406
__writeMemory32 (C-SPY system macro)	407
-d (C-SPY command line option)	423
-f (cspybat option)	428
-p (C-SPY command line option)	433
--application_args (C-SPY command line option)	420
--attach_to_running_target (C-SPY command line option)	421
--backend (C-SPY command line option)	421
--bounds_table_size (linker option)	321
--code_coverage_file (C-SPY command line option)	422
--core (C-SPY command line option)	422
--cspybat_inifile (C-SPY command line option)	423
--cycles (C-SPY command line option)	423
--debug_file (cspybat option)	424
--debug_heap (linker option)	321

--device_select (C-SPY command line option)	424
--diag_warning (C-SPY command line option)	425
--disable_interrupts (C-SPY command line option)	425
--double (C-SPY command line option)	425
--download_only (C-SPY command line option)	426
--drv_communication (C-SPY command line option)	426
--drv_mode (C-SPY command line option)	427
--endian (C-SPY command line option)	427
--flash_only_changed_blocks (C-SPY command line option)	428
--fpu (compiler option)	429
--function_profiling (cspybat option)	429
--generate_entries_without_bounds (compiler option)	321
--ignore_uninstrumented_pointers (compiler option)	322
--ignore_uninstrumented_pointers (linker option)	322
--int (C-SPY command line option)	430
--ir_length (C-SPY command line option)	430
--leave_target_running (C-SPY command line option)	431
--log_file (C-SPY command line option)	431
--macro (C-SPY command line option)	431
--macro_param (C-SPY command line option)	432
--mapu (C-SPY command line option)	432
--plugin (C-SPY command line option)	433
--rtc_enable (cspybat option)	327
--rtc_output (cspybat option)	327
--rtc_raw_to_txt (cspybat option)	328
--rtc_rules (cspybat option)	328
--runtime_checking (compiler option)	322
--set_pc_to_entry_symbol (C-SPY command line option)	434
--silent (C-SPY command line option)	434
--suppress_download (C-SPY command line option)	434
--timeout (C-SPY command line option)	435
--verify_download (C-SPY command line option)	435

Numerics

1x Units (Symbolic Memory window context menu)	168
8x Units (Memory window context menu)	161