

IAR Embedded Workbench®

C-SPY® Debugging Guide

for Advanced RISC Machines Ltd's
ARM® cores



COPYRIGHT NOTICE

© 2010–2015 IAR Systems AB.

No part of this document may be reproduced without the prior written consent of IAR Systems AB. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

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

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

EDITION NOTICE

Twelfth edition: November 2015

Part number: UCSARM-12

This guide applies to version 7.50.x of IAR Embedded Workbench® for ARM.

Internal reference: M18, Hom7.2, IMAE.

Brief contents

Tables	23
Preface	25
Part 1. Basic debugging	31
The IAR C-SPY Debugger	33
Getting started using C-SPY	49
Executing your application	71
Variables and expressions	91
Breakpoints	125
Memory and registers	161
Part 2. Analyzing your application	197
Trace	199
Profiling	255
Code coverage	269
Power debugging	275
C-RUN runtime error checking	293
Part 3. Advanced debugging	333
Multicore debugging	335
Interrupts	343
C-SPY macros	365
The C-SPY command line utility— <code>cspybat</code>	439
Flash loaders	489

Part 4. Additional reference information	495
Debugger options	497
Additional information on C-SPY drivers	541
Index	559

Contents

Tables	23
Preface	25
Who should read this guide	25
Required knowledge	25
How to use this guide	25
What this guide contains	26
Part 1. Basic debugging	26
Part 2. Analyzing your application	26
Part 3. Advanced debugging	27
Part 4. Additional reference information	27
Other documentation	27
User and reference guides	27
The online help system	28
Web sites	29
Document conventions	29
Typographic conventions	29
Naming conventions	30
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	37

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	41
The C-SPY hardware debugger drivers	41
Communication overview	41
Hardware installation	44
USB driver installation	44
Getting started using C-SPY	49
Setting up C-SPY	49
Setting up for debugging	49
Executing from reset	50
Using a setup macro file	50
Selecting a device description file	51
Loading plugin modules	51
Starting C-SPY	52
Starting a debug session	52
Loading executable files built outside of the IDE	52
Starting a debug session with source files missing	52
Loading multiple images	53
Adapting for target hardware	54
Modifying a device description file	54
Initializing target hardware before C-SPY starts	55
Remapping memory	56
Using predefined C-SPY macros for device support	57
An overview of the debugger startup	57
Debugging code in flash	58
Debugging code in RAM	59
Running example projects	59
Running an example project	59

Reference information on starting C-SPY	61
C-SPY Debugger main window	61
Images window	67
Get Alternative File dialog box	68
Get Example Projects dialog box	69
Executing your application	71
Introduction to application execution	71
Briefly about application execution	71
Source and disassembly mode debugging	71
Single stepping	72
Stepping speed	74
Running the application	75
Highlighting	76
Call stack information	76
Terminal input and output	77
Debug logging	77
Reference information on application execution	78
Disassembly window	78
Call Stack window	82
Terminal I/O window	84
Terminal I/O Log File dialog box	85
Debug Log window	86
Log File dialog box	87
Report Assert dialog box	88
Autostep settings dialog box	89
Variables and expressions	91
Introduction to working with variables and expressions	91
Briefly about working with variables and expressions	91
C-SPY expressions	92
Limitations on variable information	95
Working with variables and expressions	95
Using the windows related to variables and expressions	96
Viewing assembler variables	96

Getting started using data logging	97
Getting started using event logging	99
Reference information on working with variables and expressions	100
Auto window	101
Locals window	102
Watch window	104
Live Watch window	106
Statics window	109
Quick Watch window	112
Symbols window	114
Resolve Symbol Ambiguity dialog box	116
Data Log window	117
Data Log Summary window	119
Event Log window	121
Event Log Summary window	123
Breakpoints	125
Introduction to setting and using breakpoints	125
Reasons for using breakpoints	125
Briefly about setting breakpoints	126
Breakpoint types	126
Breakpoint icons	128
Breakpoints in the C-SPY simulator	129
Breakpoints in the C-SPY hardware debugger drivers	129
Breakpoint consumers	129
Breakpoints options	130
Setting breakpoints	130
Various ways to set a breakpoint	131
Toggling a simple code breakpoint	131
Setting breakpoints using the dialog box	131
Setting a data breakpoint in the Memory window	132
Setting breakpoints using system macros	133
Setting a breakpoint on an exception vector	134

Setting breakpoints in __ramfunc declared functions	135
Useful breakpoint hints	135
Reference information on breakpoints	137
Breakpoints window	137
Breakpoint Usage window	139
Code breakpoints dialog box	140
JTAG Watchpoints dialog box	142
Log breakpoints dialog box	145
Data breakpoints dialog box	146
Data Log breakpoints dialog box	150
Data Log breakpoints dialog box (C-SPY hardware drivers)	151
Breakpoints options dialog box	153
Immediate breakpoints dialog box	155
Vector Catch dialog box	156
Enter Location dialog box	157
Resolve Source Ambiguity dialog box	158
Memory and registers	161
Introduction to monitoring memory and registers	161
Briefly about monitoring memory and registers	161
C-SPY memory zones	163
Stack display	163
Memory access checking	164
Monitoring memory and registers	165
Configuring C-SPY to match the memory of your device	165
Defining application-specific register groups	165
Reference information on memory and registers	166
Memory window	167
Memory Save dialog box	171
Memory Restore dialog box	172
Fill dialog box	173
Symbolic Memory window	174
Stack window	177
Register window	180

SFR Setup window	183
Edit SFR dialog box	186
Memory Configuration dialog box, for the C-SPY simulator	187
Edit Memory Range dialog box, for the C-SPY simulator	190
Memory Configuration dialog box, in C-SPY hardware debugger drivers	191
Edit Memory Range dialog box, for C-SPY hardware debugger drivers	194

Part 2. Analyzing your application 197

Trace	199
Introduction to using trace	199
Reasons for using trace	199
Briefly about trace	200
Requirements for using trace	202
Collecting and using trace data	203
Getting started with ETM trace	203
Getting started with SWO trace	204
Setting up concurrent use of ETM and SWO	204
Trace data collection using breakpoints	205
Searching in trace data	205
Browsing through trace data	206
Reference information on trace	207
ETM Trace Settings dialog box	208
ETM Trace Settings dialog box (J-Link/J-Trace)	210
SWO Trace Window Settings dialog box	212
SWO Configuration dialog box	214
Trace window	218
Function Trace window	223
Timeline window	224
Viewing Range dialog box	233
Trace Start breakpoints dialog box	235
Trace Stop breakpoints dialog box	236

Trace Start breakpoints dialog box (I-jet/JTAGjet and CMSIS-DAP)	237
Trace Stop breakpoints dialog box (I-jet/JTAGjet and CMSIS-DAP)	239
Trace Filter breakpoints dialog box (I-jet/JTAGjet)	241
Trace Start breakpoints dialog box (J-Link/J-Trace)	242
Trace Stop breakpoints dialog box (J-Link/J-Trace)	245
Trace Filter breakpoints dialog box (J-Link/J-Trace)	247
Trace Expressions window	250
Find in Trace dialog box	251
Find in Trace window	252
Trace Save dialog box	253
Profiling	255
Introduction to the profiler	255
Reasons for using the profiler	255
Briefly about the profiler	255
Requirements for using the profiler	256
Using the profiler	257
Getting started using the profiler on function level	258
Analyzing the profiling data	258
Getting started using the profiler on instruction level	260
Selecting a time interval for profiling information	261
Reference information on the profiler	262
Function Profiler window	263
Code coverage	269
Introduction to code coverage	269
Reasons for using code coverage	269
Briefly about code coverage	269
Requirements and restrictions for using code coverage	269
Reference information on code coverage	270
Code Coverage window	270

Power debugging	275
Introduction to power debugging	275
Reasons for using power debugging	275
Briefly about power debugging	275
Requirements and restrictions for power debugging	276
Optimizing your source code for power consumption	277
Waiting for device status	277
Software delays	277
DMA versus polled I/O	278
Low-power mode diagnostics	278
CPU frequency	278
Detecting mistakenly unattended peripherals	279
Peripheral units in an event-driven system	279
Finding conflicting hardware setups	280
Analog interference	281
Debugging in the power domain	281
Displaying a power profile and analyzing the result	282
Detecting unexpected power usage during application execution ...	283
Changing the graph resolution	284
Reference information on power debugging	284
Power Log Setup window	285
Power Log window	287
Power graph in the Timeline window	290
C-RUN runtime error checking	293
Introduction to runtime error checking	293
Runtime error checking	293
Runtime error checking using C-RUN	294
The checked heap provided by the library	295
Using C-RUN in the IAR Embedded Workbench IDE	295
Using C-RUN in non-interactive mode	296
Requirements for runtime error checking	296
Using C-RUN	297
Getting started using C-RUN runtime error checking	297

Creating rules for messages	299
Detecting various runtime errors	299
Detecting implicit or explicit integer conversion	299
Detecting signed or unsigned overflow	301
Detecting bit loss or undefined behavior when shifting	303
Detecting division by zero	304
Detecting unhandled cases in switch statements	304
Detecting accesses outside the bounds of arrays and other objects ..	305
Detecting heap usage error	311
Detecting heap memory leaks	312
Detecting heap integrity violations	314
Reference information on runtime error checking	317
C-RUN Runtime Checking options	318
C-RUN Messages window	320
C-RUN Messages Rules window	322
Compiler and linker reference for C-RUN	324
--bounds_table_size (linker option)	324
--debug_heap (linker option)	325
--generate_entries_without_bounds	325
--ignore_uninstrumented_pointers	326
--ignore_uninstrumented_pointers (linker option)	326
--runtime_checking	326
#pragma default_no_bounds	327
#pragma define_with_bounds	327
#pragma define_without_bounds	328
#pragma disable_check	328
#pragma generate_entry_without_bounds	328
#pragma no_bounds	329
__as_get_base	329
__as_get_bound	329
__as_make_bounds	330
cspybat options for C-RUN	330
--rtc_enable	330
--rtc_output	331

--rtc_raw_to_txt	331
--rtc_rules	332
Part 3. Advanced debugging	333
Multicore debugging	335
Introduction to multicore debugging	335
Briefly about multicore debugging	335
Symmetric multicore debugging	335
Asymmetric multicore debugging	336
Requirements and restrictions for multicore debugging	337
Debugging multiple cores	337
Setting up for symmetric multicore debugging	337
Setting up for asymmetric multicore debugging	337
Starting and stopping a multicore debug session	339
Reference information on multicore debugging	339
Cores window	339
Cores toolbar	341
Interrupts	343
Introduction to interrupts	343
Briefly about interrupt logging	343
Briefly about the interrupt simulation system	344
Interrupt characteristics	345
Interrupt simulation states	345
C-SPY system macros for interrupt simulation	346
Target-adapting the interrupt simulation system	347
Using the interrupt system	347
Simulating a simple interrupt	348
Simulating an interrupt in a multi-task system	349
Getting started using interrupt logging	350
Reference information on interrupts	350
Interrupt Setup dialog box	351
Edit Interrupt dialog box	353

Forced Interrupt window	354
Interrupt Status window	355
Interrupt Log window	357
Interrupt Log Summary window	361
C-SPY macros	365
Introduction to C-SPY macros	365
Reasons for using C-SPY macros	365
Briefly about using C-SPY macros	366
Briefly about setup macro functions and files	366
Briefly about the macro language	367
Using C-SPY macros	367
Registering C-SPY macros—an overview	368
Executing C-SPY macros—an overview	368
Registering and executing using setup macros and setup files	369
Executing macros using Quick Watch	370
Executing a macro by connecting it to a breakpoint	370
Aborting a C-SPY macro	372
Reference information on the macro language	372
Macro functions	372
Macro variables	373
Macro parameters	373
Macro strings	374
Macro statements	374
Formatted output	375
Reference information on reserved setup macro function names	377
execUserPreload	377
execUserExecutionStarted	378
execUserExecutionStopped	378
execUserFlashInit	378
execUserSetup	378
execUserFlashReset	379
execUserPreReset	379

execUserReset	379
execUserExit	380
execUserFlashExit	380
Reference information on C-SPY system macros	380
__cancelAllInterrupts	383
__cancelInterrupt	384
__clearBreak	384
__closeFile	384
__delay	385
__disableInterrupts	385
__driverType	385
__emulatorSpeed	386
__emulatorStatusCheckOnRead	387
__enableInterrupts	388
__evaluate	388
__fillMemory8	389
__fillMemory16	389
__fillMemory32	390
__gdbserver_exec_command	391
__getSelectedCore	392
__getTracePortSize	392
__hasDAPRegs	393
__hwJetResetWithStrategy	393
__hwReset	394
__hwResetRunToBp	394
__hwResetWithStrategy	396
__hwRunToBreakpoint	396
__isBatchMode	397
__jlinkExecCommand	398
__jtagCommand	398
__jtagCP15IsPresent	399
__jtagCP15ReadReg	399
__jtagCP15WriteReg	399
__jtagData	400

__jtagRawRead	400
__jtagRawSync	401
__jtagRawWrite	402
__jtagResetTRST	403
__loadImage	403
__memoryRestore	404
__memorySave	405
__messageBoxYesCancel	406
__messageBoxYesNo	406
__openFile	407
__orderInterrupt	408
__popSimulatorInterruptExecutingStack	409
__readAPReg	409
__readDPRreg	410
__readFile	411
__readFileByte	411
__readMemory8, __readMemoryByte	412
__readMemory16	412
__readMemory32	413
__registerMacroFile	413
__resetFile	414
__restoreSoftwareBreakpoints	414
__selectCore	415
__setCodeBreak	415
__setDataBreak	416
__setDataLogBreak	418
__setLogBreak	419
__setSimBreak	420
__setTraceStartBreak	421
__setTraceStopBreak	423
__sourcePosition	425
__strFind	425
__subString	426
__targetDebuggerVersion	426

__toLower	427
__toString	427
__toUpper	428
__unloadImage	428
__writeAPReg	429
__writeDPReg	429
__writeFile	430
__writeFileByte	430
__writeMemory8, __writeMemoryByte	431
__writeMemory16	431
__writeMemory32	432
Graphical environment for macros	432
Macro Registration window	433
Debugger Macros window	435
Macro Quicklaunch window	437
The C-SPY command line utility—cspybat	439
Using C-SPY in batch mode	439
Starting cspybat	439
Output	440
Invocation syntax	440
Summary of C-SPY command line options	441
General cspybat options	441
Options available for all C-SPY drivers	442
Options available for the simulator driver	444
Options available for the C-SPY Angel debug monitor driver	444
Options available for the C-SPY GDB Server driver	444
Options available for the C-SPY IAR ROM-monitor driver	444
Options available for the C-SPY I-jet/JTAGjet driver	444
Options available for the C-SPY CMSIS-DAP driver	445
Options available for the C-SPY J-Link/J-Trace driver	446
Options available for the C-SPY TI Stellaris driver	447
Options available for the C-SPY TI XDS driver	447
Options available for the C-SPY Macraigor driver	447

Options available for the C-SPY RDI driver	448
Options available for the C-SPY ST-LINK driver	448
Options available for the C-SPY third-party drivers	449
Reference information on C-SPY command line options ...	449
--backend	449
--code_coverage_file	449
--cycles	450
--debugfile	450
--device	451
--disable_interrupts	451
--download_only	451
--drv_catch_exceptions	452
--drv_communication	453
--drv_communication_log	459
--drv_default_breakpoint	459
--drv_interface	460
--drv_interface_speed	461
--drv_reset_to_cpu_start	462
--drv_restore_breakpoints	462
--drv_swo_clock_setup	463
--drv_vector_table_base	464
-f	465
--flash_loader	465
--gdbserv_exec_command	466
--jet_board_cfg	466
--jet_board_did	466
--jet_cpu_clock	467
--jet_ir_length	468
--jet_power_from_probe	468
--jet_probe	469
--jet_script_file	469
--jet_standard_reset	470
--jet_startup_connection_timeout	471
--jet_swo_on_d0	471

--jet_swo_prescaler	472
--jet_swo_protocol	472
--jet_tap_position	473
--jlink_dcc_timeout	473
--jlink_device_select	474
--jlink_exec_command	474
--jlink_initial_speed	474
--jlink_ir_length	475
--jlink_reset_strategy	475
--jlink_script_file	476
--jlink_trace_source	476
--leave_running	477
--macro	477
--macro_param	477
--mac_handler_address	478
--mac_jtag_device	478
--mac_multiple_targets	479
--mac_reset_pulls_reset	479
--mac_set_temp_reg_buffer	480
--mac_xscale_ir7	480
--mapu	480
-p	481
--plugin	481
--proc_stack_stack	482
--rdi_allow_hardware_reset	482
--rdi_driver_dll	483
--rdi_step_max_one	483
--reset_style	483
--semihosting	484
--silent	485
--stlink_reset_strategy	485
--timeout	486
--xds_board_file	486
--xds_reset_strategy	487

--xds_rootdir	487
Flash loaders	489
Introduction to the flash loader	489
Using flash loaders	489
Setting up the flash loader(s)	489
The flash loading mechanism	490
Aborting a flash loader	490
Reference information on the flash loader	491
Flash Loader Overview dialog box	491
Flash Loader Configuration dialog box	493
Part 4. Additional reference information	495
Debugger options	497
Setting debugger options	497
Reference information on general debugger options	498
Setup	499
Download	500
Images	501
Plugins	502
Extra Options	503
Multicore	504
Reference information on C-SPY hardware debugger driver options	505
Angel	506
Setup options for CMSIS-DAP	507
JTAG/SWD options for CMSIS-DAP	510
GDB Server	512
IAR ROM-monitor	513
Setup options for I-jet/JTAGjet	514
JTAG/SWD options for I-jet/JTAGjet	517
Trace options for I-jet/JTAGjet	519
Setup options for J-Link/J-Trace	523

Connection options for J-Link/J-Trace	528
Macraigor	530
RDI	532
Setup options for ST-LINK	533
Communication options for ST-LINK	535
Setup options for TI Stellaris	536
Setup options for TI XDS	537
Communication options for TI XDS	538
Third-Party Driver options	539
Additional information on C-SPY drivers	541
Reference information on C-SPY driver menus	541
<i>C-SPY driver</i>	541
Simulator menu	542
CMSIS-DAP menu	544
GDB Server menu	545
I-jet/JTAGjet menu	546
J-Link menu	549
Macraigor JTAG menu	552
RDI menu	552
ST-LINK menu	553
TI Stellaris menu	554
TI XDS menu	554
Reference information on the C-SPY simulator	554
Simulated Frequency dialog box	555
Resolving problems	555
No contact with the target hardware	556
Slow stepping speed	556
Index	559

Tables

1: Typographic conventions used in this guide	29
2: Naming conventions used in this guide	30
3: Driver differences, I-jet/JTAGjet, J-Link/J-Trace and ST-LINK	39
4: Driver differences, other drivers	40
5: Terminal I/O in real time	84
6: C-SPY assembler symbols expressions	93
7: Handling name conflicts between hardware registers and assembler labels	94
8: Live watch for the different devices	106
9: C-SPY macros for breakpoints	133
10: C-SPY macros for breakpoints	133
11: Supported graphs in the Timeline window	225
12: C-SPY driver profiling support	257
13: Project options for enabling the profiler	258
14: Project options for enabling code coverage	271
15: Timer interrupt settings	349
16: Examples of C-SPY macro variables	373
17: Summary of system macros	380
18: __cancelInterrupt return values	384
19: __disableInterrupts return values	385
20: __driverType return values	386
21: __emulatorSpeed return values	387
22: __enableInterrupts return values	388
23: __evaluate return values	388
24: __getTracePortSize return values	392
25: __hasDAPRegs return values	393
26: __hwJetResetWithStrategy return values	393
27: __hwReset return values	394
28: __hwResetRunToBp return values	395
29: __hwResetWithStrategy return values	396
30: __hwRunToBreakpoint return values	397
31: __isBatchMode return values	397

32: __jtagResetTRST return values	403
33: __loadImage return values	404
34: __messageBoxYesCancel return values	406
35: __messageBoxYesNo return values	407
36: __openFile return values	407
37: __readAPReg return values	410
38: __readDPReg return values	410
39: __readFile return values	411
40: __setCodeBreak return values	416
41: __setDataBreak return values	418
42: __setDataLogBreak return values	419
43: __setLogBreak return values	420
44: __setSimBreak return values	421
45: __setTraceStartBreak return values	422
46: __setTraceStopBreak return values	424
47: __sourcePosition return values	425
48: __unloadImage return values	429
49: __writeAPReg return values	429
50: __writeDPReg return values	430
51: cspybat parameters	440
52: Options specific to the C-SPY drivers you are using	497
53: Catching exceptions	533

Preface

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

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 ARM core (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 27.

How to use this guide

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

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

This guide describes a number of *topics*, where each topic section contains an introduction which also covers concepts related to the topic. This will give you a good understanding of the features in C-SPY. Furthermore, the topic section provides

procedures with step-by-step descriptions to help you use the features. Finally, each topic section gives all relevant reference information.

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

What this guide contains

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 ARM.

PART I. BASIC DEBUGGING

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

PART 2. ANALYZING YOUR APPLICATION

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

- *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

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

PART 4. ADDITIONAL REFERENCE INFORMATION

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

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in the IAR Embedded Workbench IDE. The online help system is also available via the F1 key.

USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, is available in the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.

- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for ARM*.
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for ARM*.
- Programming for the IAR C/C++ Compiler for ARM and linking using the IAR ILINK Linker, is available in the *IAR C/C++ Development Guide for ARM*.
- Programming for the IAR Assembler for ARM, is available in the *IAR Assembler Reference Guide for ARM*.
- Using the IAR DLIB C/C++ standard library functions, is available in the *DLIB C/C++ standard library reference*, available in the online help system.
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Developing safety-critical applications using the MISRA C guidelines, is available in the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
- Using I-jet, refer to the *IAR Debug probes User Guide for I-jet®, I-jet Trace, and I-scope*.
- Using JTAGjet-Trace, refer to the *JTAGjet-Trace User Guide for ARM*.
- Using IAR J-Link and IAR J-Trace, refer to the *IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for ARM, is available in the *IAR Embedded Workbench® Migration Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

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

WEB SITES

Recommended web sites:

- The Advanced RISC Machines Ltd web site, www.arm.com, that contains information and news about the ARM cores.
- The IAR Systems web site, www.iar.com, that holds application notes and other product information.
- The web site of the C standardization working group, www.open-std.org/jtc1/sc22/wg14.
- The web site of the C++ Standards Committee, www.open-std.org/jtc1/sc22/wg21.
- Finally, the Embedded C++ Technical Committee web site, www.caravan.net/ec2plus, that contains information about the Embedded C++ standard.

Document conventions

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

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

TYPOGRAPHIC CONVENTIONS

The IAR Systems 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 directive, where [and] are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
{option}	A mandatory part of a directive, where { and } are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
[option]	An optional part of a command.
[a b c]	An optional part of a command with alternatives.
{a b c}	A mandatory part of a command with alternatives.

Table 1: Typographic conventions used in this guide





Style	Used for
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 Systems®, when referred to in the documentation:

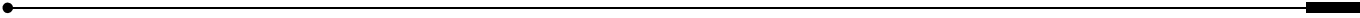
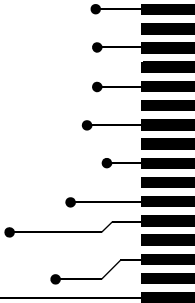
Brand name	Generic term
IAR Embedded Workbench® for ARM	IAR Embedded Workbench®
IAR Embedded Workbench® IDE for ARM	the IDE
IAR C-SPY® Debugger for ARM	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for ARM	the compiler
IAR Assembler™ for ARM	the assembler
IAR ILINK Linker™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

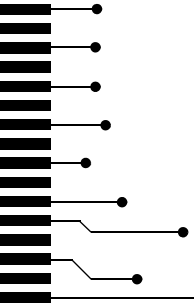
Table 2: Naming conventions used in this guide

Part I. Basic debugging

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

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





The IAR C-SPY Debugger

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

Introduction to C-SPY

These topics are covered:

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

AN INTEGRATED ENVIRONMENT

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

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

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

GENERAL C-SPY DEBUGGER FEATURES

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

C-SPY offers these general features:

- Source and disassembly level debugging

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

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

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

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

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

The compiler generates extensive call stack information. This allows the debugger to show, without any runtime penalty, the complete stack of function calls wherever the program counter is. You can select any function in the call stack, and for each function you get valid information for local variables and available registers.
- Powerful macro system

C-SPY includes a powerful internal macro system, to allow you to define complex sets of actions to be performed. C-SPY macros can be used on their own or in

conjunction with complex breakpoints and—if you are using the simulator—the interrupt simulation system to perform a wide variety of tasks.

Additional general C-SPY debugger features

This list shows some additional features:

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

RTOS AWARENESS

C-SPY supports RTOS-aware debugging.

These operating systems are currently supported:

- AVIX-RT
- CMX-RTX
- CMX-Tiny+
- eForce mC3/Compact
- eSysTech X realtime kernel
- Express Logic ThreadX
- FreeRTOS, OpenRTOS, and SafeRTOS
- Freescale MQX
- Micrium uC/OS-II
- Micrium uC/OS-III
- Micro Digital SMX
- MISPO NORTi
- OSEK Run Time Interface (ORTI)
- RTX C Quadros
- Segger embOS
- unicon Fusion.

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

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

Debugger concepts

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

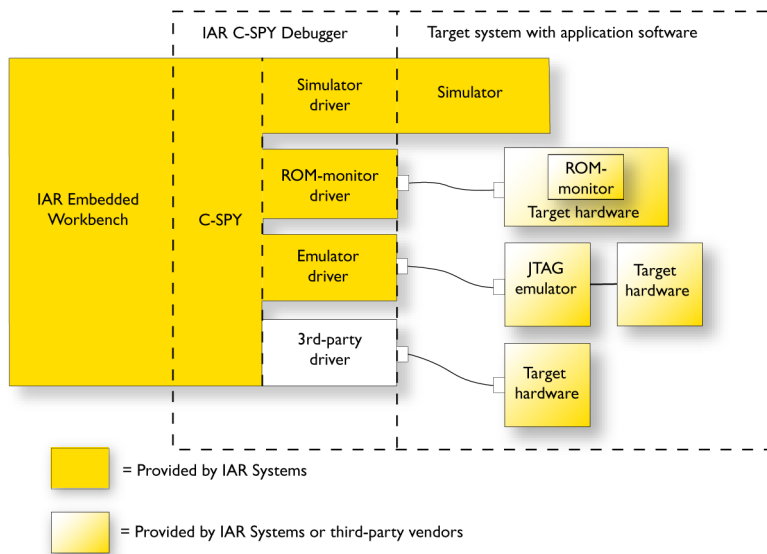
These topics are covered:

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

C-SPY AND TARGET SYSTEMS

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

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



THE DEBUGGER

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

THE TARGET SYSTEM

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

THE APPLICATION

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

C-SPY DEBUGGER SYSTEMS

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

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

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

C-SPY is available with a simulator driver, and depending on your product package, optional drivers for hardware debugger systems. For an overview of the available C-SPY drivers and the functionality provided by each driver, see *C-SPY drivers overview*, page 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 Systems toolchain as long as the third-party debugger can read ELF/DWARF, Intel-extended, or Motorola. For information about which format to use with a third-party debugger, see the user documentation supplied with that tool.

C-SPY PLUGIN MODULES

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

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

- Code Coverage, which is integrated in the IDE.
- The various C-SPY drivers for debugging using certain debug systems.
- RTOS plugin modules for support for real-time OS aware debugging.
- Peripheral simulation modules make C-SPY simulate peripheral units. Such plugin modules are not provided by IAR Systems, but can be developed and distributed by third-party suppliers.
- C-SPYLink that bridges IAR visualSTATE and IAR Embedded Workbench to make true high-level state machine debugging possible directly in C-SPY, in addition to

the normal C level symbolic debugging. For more information, see the documentation provided with IAR visualSTATE.

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

C-SPY drivers overview

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

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

Note: In addition to the drivers supplied with IAR Embedded Workbench, you can also load debugger drivers supplied by a third-party vendor; see *Third-Party Driver options*, page 539.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

This table summarizes the key differences between the Simulator, I-jet/JTAGjet, J-Link/J-Trace, ST-LINK, and CMSIS-DAP:

Feature	Simulator	I-jet/JTAGjet	J-Link/J-Trace	ST-LINK	CMSIS-DAP
Code breakpoints	x	x	x	x	x
Data breakpoints	x	x	x	x	x
Interrupt logging ¹	x	x	x	x	--

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

Feature	Simulator	I-jet/JTAGjet	J-Link/J-Trace	ST-LINK	CMSIS-DAP
Data logging ¹	x	x	x	x	--
Call stack trace ¹	x	x	x	--	x
Event logging ¹	--	x	x	--	--
Live watch ¹	--	x	x	x	x
Cycle counter ¹	x	x	x	x	x
Code coverage ¹	x	x	x	x	x
Data coverage	x	--	--	--	--
Function /instruction profiler ¹	x	x	x	x	x
Trace ¹	x	x	x	x	x
Multicore debugging ¹	x	x	--	--	x ²
Power debugging	--	x	x	--	--

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

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

2 Limited support.

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

Feature	Simulator	RDI	Mac- raigor	GDB Server	TI Stellaris	TI XDS	Angel
Code breakpoints	x	x	x	x	x	x	x
Data breakpoints	x	x	x	x	x	--	--
Interrupt logging	x	--	--	--	--	--	--
Cycle counter	x	--	--	--	--	--	--
Code coverage	x	--	--	--	--	--	--
Data coverage	x	--	--	--	--	--	--
Function/instruction profiler	x	--	--	--	--	--	--
Trace ¹	x	--	--	--	--	--	--

Table 4: Driver differences, other drivers

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

The IAR C-SPY Simulator

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

The C-SPY Simulator supports:

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

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

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

The C-SPY hardware debugger drivers

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

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

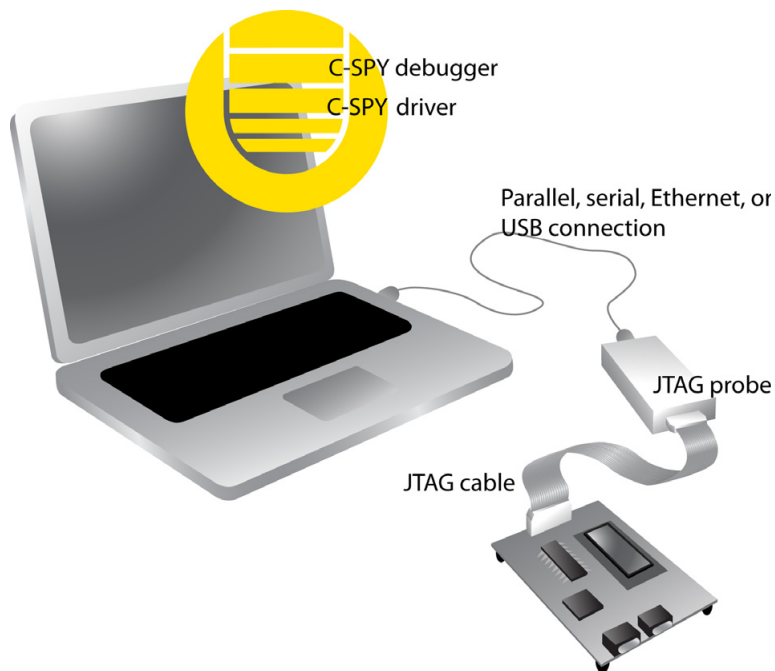
COMMUNICATION OVERVIEW

There are two main communication setups, depending on the type of target system. Many of the arm cores have built-in, on-chip debug support. Because the hardware

debugger logic is built into the core, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work, other than the debug probe. For some devices that do not have such built-in, on-chip debug support, there are instead a ROM-monitor debugger solution that can be used.

Overview of a target system with a debug probe or emulator

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



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

Overview of a target system using a ROM-monitor

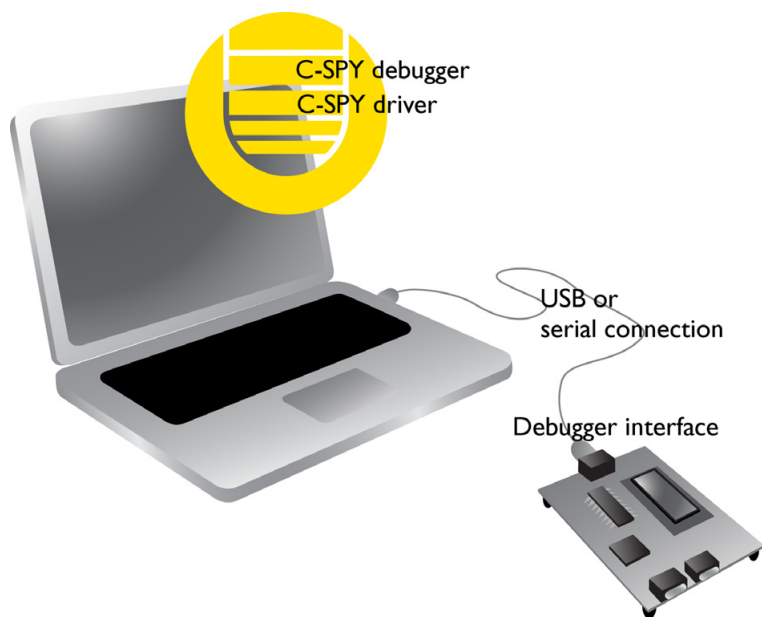
IAR Embedded Workbench comes with two ready-made ROM-monitors:

- Using the IAR Angel debug monitor driver, you can communicate with any device compliant with the Angel debug monitor protocol. In most cases these are evaluation boards.

- Using the IAR ROM-monitor driver, C-SPY can connect to the Analog Devices ADuC7xxx boards and the IAR Kickstart Card for Philips LPC210x. Most ROM-monitors require that the code you want to debug is located in RAM, because the only way you can set breakpoints and step in your application code is to download it to RAM. For some ROM-monitors, for example for Analog Devices ADuC7xxx, the code that you want to debug can be located in flash memory. To maintain debug functionality, the ROM-monitor might simulate some instructions, for example when single stepping.

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

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



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

For further information, see:

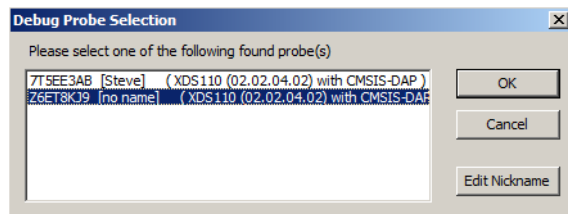
- The `angel_quickstart.html` file, available in the `arm\doc\infocenter` directory, or refer to the manufacturer's documentation.
- The `iar_rom_quickstart.html` file, available in the `arm\doc\infocenter` directory, or refer to the manufacturer's documentation.

HARDWARE INSTALLATION

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

- 1 Connect the probe to the target board.
- 2 Connect the USB cable to the debug probe.
- 3 Power up the debug probe, if it is not powered via USB.
- 4 Power up the target board, if it is not powered by the debug probe.
- 5 Start the C-SPY debugging session.
- 6 If more than one debug probe is connected to your computer, the **Debug Probe Selection** dialog box is displayed. In the dialog box, select the probe to use and click **OK**. For more information, see *--drv_communication*, page 453.

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



USB DRIVER INSTALLATION

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

Installing the I-jet and JTAGjet USB driver

Before you can use the I-jet or the JTAGjet interface over the USB port, the proper USB driver must be installed. Use the USB cable to connect the computer to the I-jet, JTAGjet, or JTAGjet-Trace probe.

Windows 7 and later

- 1 Start the Windows Device Manager.
- 2 Select **Other devices**, right-click on **JTAGjet** and select **Update Driver Software**.
- 3 Click **Browse my computer for driver software** and browse to the `arm\drivers\jet\USB`.
- 4 Click **Next** and then **Install**.

Before Windows 7

The first time that the I-jet or JTAGjet interface and the computer are connected, Windows opens a dialog box and asks you to locate the USB driver. The drivers can be found in the product installation in the `arm\drivers\jet\USB`.

Once the initial setup is completed, you do not need to install the driver again.

Installing the J-Link USB driver

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

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

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

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

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

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

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

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

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

`ST-Link_V2_USBdriver.exe`.

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

Installing the TI Stellaris USB driver

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

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

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

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

Installing the TI XDS USB driver

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

- 1 Install IAR Embedded Workbench for ARM.
- 2 Install the TI XDS package which can be found in the `arm\drivers\ti-xds` directory. It is recommended to choose the suggested installation directory. See also *Setup options for TI XDS*, page 537.
- 3 Use the USB cable to connect the computer to the TI board.

Configuring the OpenOCD Server

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

Getting started using C-SPY

- Setting up C-SPY
- Starting C-SPY
- Adapting for target hardware
- An overview of the debugger startup
- Running example projects
- Reference information on starting C-SPY

Setting up C-SPY

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

- I Install a USB driver or some other communication driver if your C-SPY driver requires it.

For more information, see:

- *Installing the I-jet and JTAGjet USB driver*, page 45
- *Installing the J-Link USB driver*, page 45
- *Installing the ST-LINK USB driver for ST-LINK ver. 2*, page 46
- *Installing the TI Stellaris USB driver*, page 46
- *Installing the TI XDS USB driver*, page 46
- *Configuring the OpenOCD Server*, page 47

- 2 Before you start C-SPY, choose **Project>Options>Debugger>Setup** and select the C-SPY driver that matches your debugger system: simulator or a hardware debugger system.
- 3 In the **Category** list, select the appropriate C-SPY driver and make your settings.
For information about these options, see *Debugger options*, page 497.
- 4 Click **OK**.
- 5 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 ARM*.

See also *Adapting for target hardware*, page 54.

EXECUTING FROM RESET

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

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

If you leave the check box empty, the program counter will 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 365. For an example of how to use a setup macro file, see *Initializing target hardware before C-SPY starts*, page 55.

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—either an IAR-specific `ddf` file or a CMSIS System View 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 Systems are provided in the `arm\config` directory and they have the filename extension `ddf`.

For more information about device description files, see *Adapting for target hardware*, page 54.

To override the default device description file:

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

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

LOADING PLUGIN MODULES

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

For more information, see *Plugins*, page 502.

Starting C-SPY

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

These tasks are covered:

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

STARTING A DEBUG SESSION

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



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



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

LOADING EXECUTABLE FILES BUILT OUTSIDE OF THE IDE

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

To create a project for an externally built file:

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

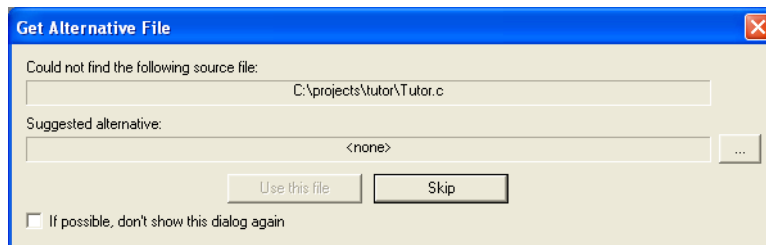


The only project options that are meaningful to set for this kind of project are options in the **General Options** and **Debugger** categories. Make sure to set up the general project options in the same way as when the executable file was built.

STARTING A DEBUG SESSION WITH SOURCE FILES MISSING

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

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



Typically, you can use the dialog box like this:

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

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

For more information, see *Get Alternative File dialog box*, page 68.

LOADING MULTIPLE IMAGES

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

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

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

To load additional images at C-SPY startup:

- 1 Choose **Project>Options>Debugger>Images** and specify up to three additional images to be loaded. For more information, see *Images*, page 501.
- 2 Start the debug session.

To load additional images at a specific moment:

Use the `__loadImage` system macro and execute it using either one of the methods described in *Using C-SPY macros*, page 367.

To display a list of loaded images:

Choose **Images** from the **View** menu. The **Images** window is displayed, see *Images window*, page 67.

Adapting for target hardware

These tasks are covered:

- Modifying a device description file
- Initializing target hardware before C-SPY starts
- Remapping memory
- Using predefined C-SPY macros for device support

See also *Configuring C-SPY to match the memory of your device*, page 165.

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 51. They contain device-specific information such as:

- Definitions of registers in peripheral units and groups of these.
- Interrupt definitions (for Cortex-M devices only); see *Interrupts*, page 343.

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

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



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

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

For information about how to load a device description file, see *Selecting a device description file*, page 51.

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 option **Use Setup file** and choose the macro file you just created.

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

REMAPPING MEMORY

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

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

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

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

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

For instructions on how to install a macro file in C-SPY, see *Registering and executing using setup macros and setup files*, page 369. For information about the macro functions used, see *Reference information on C-SPY system macros*, page 380.

USING PREDEFINED C-SPY MACROS FOR DEVICE SUPPORT

For some ARM devices, there are predefined C-SPY macros available for specific device support, typically provided by the chip manufacturer. These macros are useful for performing certain device-specific tasks,

You can easily access and execute these macros using the **Macro Quicklaunch** window.

An overview of the debugger startup

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

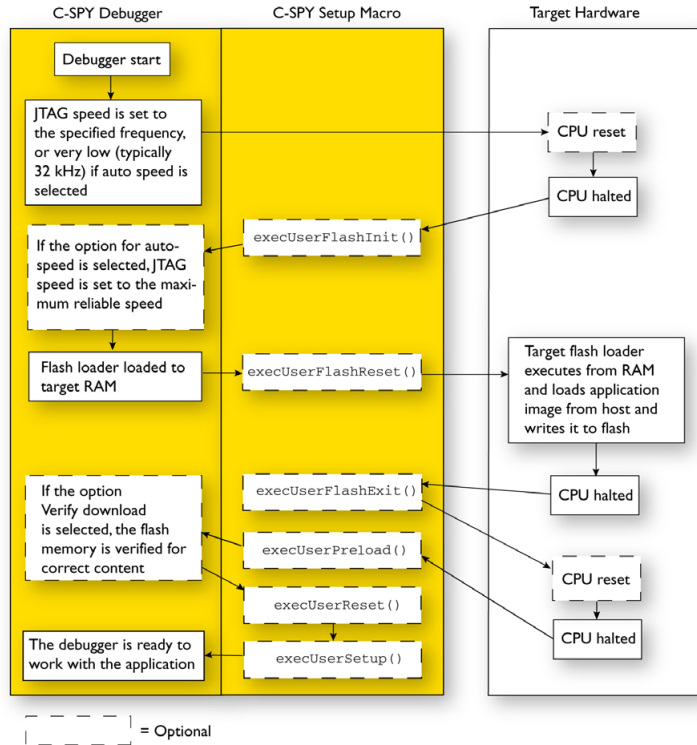
These topics are covered:

- Debugging code in flash
- Debugging code in RAM

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

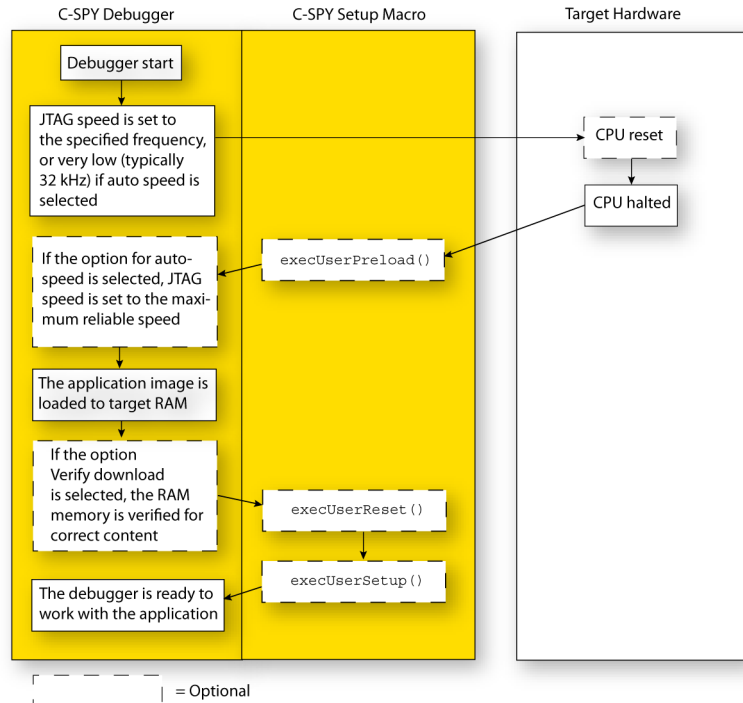
DEBUGGING CODE IN FLASH

This figure illustrates the debugger startup when debugging code in flash memory:



DEBUGGING CODE IN RAM

This figure illustrates the debugger startup when debugging code in RAM:



Running example projects

These tasks are covered:

- Running an example project

RUNNING AN EXAMPLE PROJECT

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

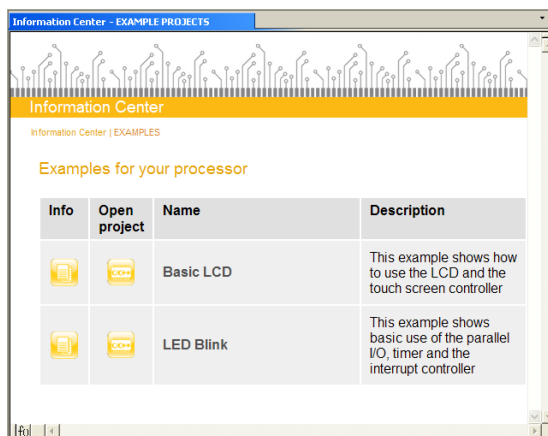
The examples are ready to be used as is. They are supplied with ready-made workspace files, together with source code files and all other related files.

To run an example project:

- 1 Choose **Help>Information Center** and click **EXAMPLE PROJECTS**.
- 2 Click the download button for the chip manufacturer that matches your device.
- 3 In the dialog box that is displayed, choose where to get the examples from. Choose between:
 - Download from IAR Systems
 - Copy from the installation DVD. In this case, use the browse button to locate the required self extracting example archive. You can find the archive in the `\examples-archive` directory on the DVD.

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

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



- 5 Click the **Open Project** button.
- 6 In the dialog box that appears, choose a destination folder for your project.
- 7 The available example projects are displayed in the workspace window. Select one of the projects, and if it is not the active project (highlighted in bold), right-click it and choose **Set As Active** from the context menu.

- 8** To view the project settings, select the project and choose **Options** from the context menu. Verify the settings for **General Options>Target>Processor variant** and **Debugger>Setup>Driver**. As for other settings, the project is set up to suit the target system you selected.

For more information about the C-SPY options and how to configure C-SPY to interact with the target board, see *Debugger options*, page 497.

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



- 9** To compile and link the application, choose **Project>Make** or click the **Make** button.
- 10** To start C-SPY, choose **Project>Debug** or click the **Download and Debug** button. If C-SPY fails to establish contact with the target system, see *Resolving problems*, page 555.



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

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

Reference information on starting C-SPY

Reference information about:

- *C-SPY Debugger main window*, page 61
- *Images window*, page 67
- *Get Alternative File dialog box*, page 68
- *Get Example Projects dialog box*, page 69

See also:

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

C-SPY Debugger main window

When you start 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 541.

Disassembly

Provides commands for executing and debugging the source application.

Debug menu

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



These commands are available:



Go F5

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

**Break**

Stops the application execution.

**Reset**

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

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

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

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

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

**Step Over (F10)**

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

**Step Into (F11)**

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

**Step Out (Shift+F11)**

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

**Next Statement**

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

**Run to Cursor**

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

Autostep

Displays a dialog box where you can customize and perform autosteping, see *Autostep settings dialog box*, page 89.

Set Next Statement

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

C++ Exceptions>

Break on Throw

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

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

C++ Exceptions>

Break on Uncaught Exception

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

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

Memory>Save

Displays a dialog box where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 171.

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 172.

Refresh

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

Macros

Displays a dialog box where you can list, register, and edit your macro files and functions, see *Using C-SPY macros*, page 367.

Logging>Set Log file

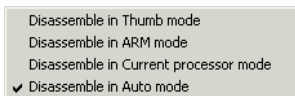
Displays a dialog box where you can choose to log the contents of the **Debug Log** window to a file. You can select the type and the location of the log file. You can choose what you want to log: errors, warnings, system information, user messages, or all of these. See *Log File dialog box*, page 87.

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 85

Disassembly menu

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



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

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

These commands are available:

Disassemble in Thumb mode Disassembles your application in Thumb mode.

Disassemble in ARM mode Disassembles your application in ARM mode.

Disassemble in Current processor mode Disassembles your application in the current processor mode.

Disassemble in Auto mode Disassembles your application in automatic mode. This is the default option.

See also *Disassembly window*, page 78.

C-SPY windows

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

- C-SPY Debugger main window

- Disassembly window
- Memory window
- Symbolic Memory window
- Register window
- Watch window
- Locals window
- Auto window
- Live Watch window
- Quick Watch window
- Statics window
- Call Stack window
- Trace window
- Function Trace window
- Timeline window
- Terminal I/O window
- Code Coverage window
- Function Profiler window
- Images window
- Stack window
- Symbols window.

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

Editing in C-SPY windows

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

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

Enter Makes an item editable and saves the new value.

Esc Cancels a new value.

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

```
myArray;3
```

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

```
myPtr;3
```

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

```
myArray;5,10
```

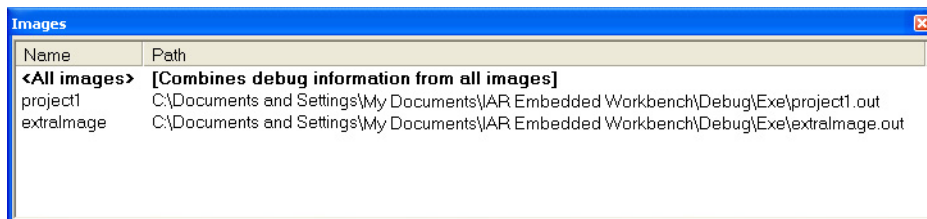
To display `myPtr+10`, `myPtr+11`, `myPtr+12`, `myPtr+13`, and `myPtr+14`, write:

```
myPtr;5,10
```

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

Images window

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



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

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

Requirements

None; this window is always available.

Display area

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

This area lists the loaded images in these columns:

Name

The name of the loaded image.

Path

The path to the loaded image.

Context menu

This context menu is available:



These commands are available:

Show all images

Shows debug information for all loaded debug images.

Show only *image*

Shows debug information for the selected debug image.

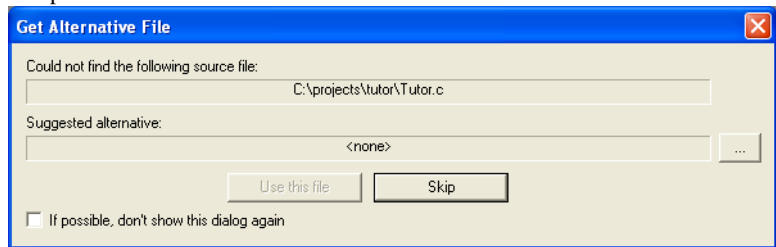
Related information

For related information, see:

- *Loading multiple images*, page 53
- *Images*, page 501
- *__loadImage*, page 403.

Get Alternative File dialog box

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



Could not find the following source file

The missing source file.

Suggested alternative

Specify an alternative file.

Use this file

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

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

Skip

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

If possible, don't show this dialog again

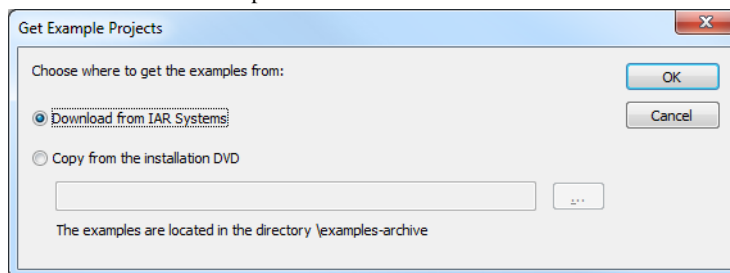
Instead of displaying the dialog box again for a missing source file, C-SPY will use the previously supplied response.

Related information

For related information, see *Starting a debug session with source files missing*, page 52.

Get Example Projects dialog box

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



See also, *Running example projects*, page 59.

Download from IAR Systems

Downloads the application from IAR Systems.

Copy from the installation DVD

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

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

Executing your application

- Introduction to application execution
- Reference information on application execution

Introduction to application execution

These topics are covered:

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

BRIEFLY ABOUT APPLICATION EXECUTION

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

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

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

SOURCE AND DISASSEMBLY MODE DEBUGGING

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

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

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

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

SINGLE STEPPING

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

There are several factors that can slow down the stepping speed. If you find it too slow, see *Slow stepping speed*, page 556 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 89.

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.

STEPPING SPEED

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

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

For this reason, it can be helpful to keep track of how many hardware breakpoints are used and make sure to some of them are left for stepping. For more information, see and *Breakpoint consumers*, page 129.

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

- If Trace or Function profiling is enabled. This might slow down stepping because collected Trace data is processed after each step. Note that it is not sufficient to close the corresponding windows to disable Trace data collection. Instead, you must disable the **Enable/Disable** button in both the Trace and the Function profiling windows.
- If the **Register** window is open and displays SFR registers. This might slow down stepping because all registers in the selected register group must be read from the hardware after each step. To solve this, you can choose to view only a limited selection of SFR register; you can choose between two alternatives. Either type `#SFR_name` (where `#SFR_name` reflects the name of the SFR you want to monitor) in the **Watch** window, or create your own filter for displaying a limited group of SFRs in the **Register** window. See *Defining application-specific register groups*, page 165.
- If any of the **Memory** or **Symbolic** memory windows is open. This might slow down stepping because the visible memory must be read after each step.
- If any of the expression related windows such as **Watch**, **Live Watch**, **Locals**, **Statics** is open. This might slow down stepping speed because all these windows reads memory after each step.
- If the **Stack** window is open and especially if the option **Enable graphical stack display and stack usage tracking** option is enabled. To disable this option, choose **Tools>Options>Stack** and disable it.
- If a too slow communication speed has been set up between C-SPY and the target board/emulator you should consider to increase the speed, if possible.

RUNNING THE APPLICATION



Go

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



Run to Cursor

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

HIGHLIGHTING

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

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

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

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

CALL STACK INFORMATION

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



Typically, this is useful for two purposes:

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

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

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

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

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

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

TERMINAL INPUT AND OUTPUT

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



This facility is useful in two different contexts:

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

For more information, see *Terminal I/O window*, page 84 and *Terminal I/O Log File dialog box*, page 85.

DEBUG LOGGING

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



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

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

Reference information on application execution

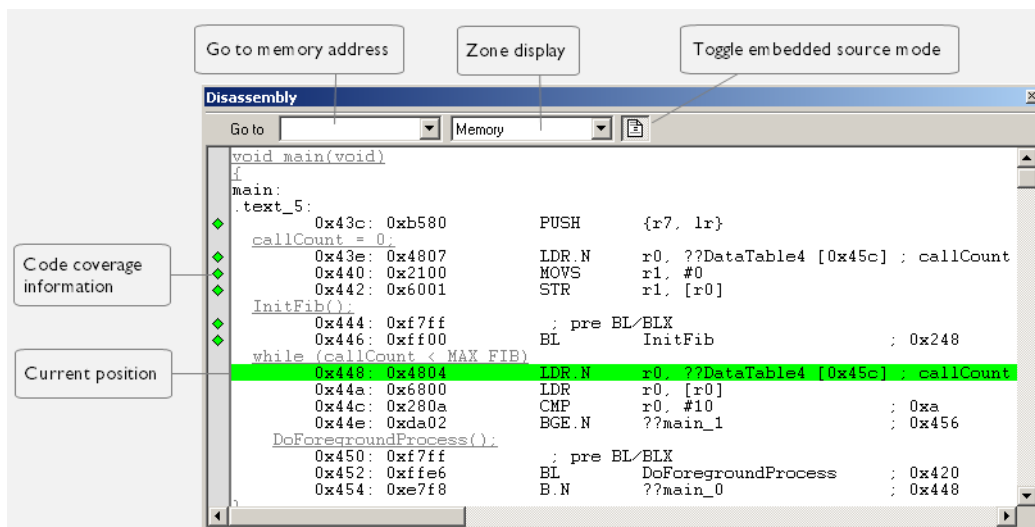
Reference information about:

- *Disassembly window*, page 78
- *Call Stack window*, page 82
- *Terminal I/O window*, page 84
- *Terminal I/O Log File dialog box*, page 85
- *Debug Log window*, page 86
- *Log File dialog box*, page 87
- *Report Assert dialog box*, page 88
- *Autostep settings dialog box*, page 89

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

Disassembly window

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



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

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

- 1 Choose **Tools>Options>Debugger**.

- 2 Set the default color using the **Source code coloring in disassembly window** option.



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

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Toggle Mixed-Mode

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

Display area

The display area shows the disassembled application code.

This area contains these graphic elements:

Green highlight	Indicates the current position, that is the next assembler instruction to be executed. To move the cursor to any line in the Disassembly window, click the line. Alternatively, move the cursor using the navigation keys.
Yellow highlight	Indicates a position other than the current position, such as when navigating between frames in the Call Stack window or between items in the Trace window.
Red dot	Indicates a breakpoint. Double-click in the gray left-side margin of the window to set a breakpoint. For more information, see <i>Breakpoints</i> , page 125.
Green diamond	Indicates code that has been executed—that is, code coverage.

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

Context menu

This context menu is available:

Move to PC	
Run to Cursor	
Code Coverage	▶
Instruction Profiling	▶
Toggle Breakpoint (Code)	
Toggle Breakpoint (Log)	
Toggle Breakpoint (Trace Start)	
Toggle Breakpoint (Trace Stop)	
Enable/disable Breakpoint	
Set Next Statement	
Copy Window Contents	
✓ Mixed-Mode	

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

These commands are available:

Move to PC

Displays code at the current program counter location.

Run to Cursor

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

Code Coverage

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

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

Instruction Profiling

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

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 140.

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 145.

Toggle Breakpoint (Trace Start)

Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Start breakpoints dialog box*, page 235.

Toggle Breakpoint (Trace Stop)

Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Stop breakpoints dialog box*, page 236.

Enable/Disable Breakpoint

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

Edit Breakpoint

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

Set Next Statement

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

Copy Window Contents

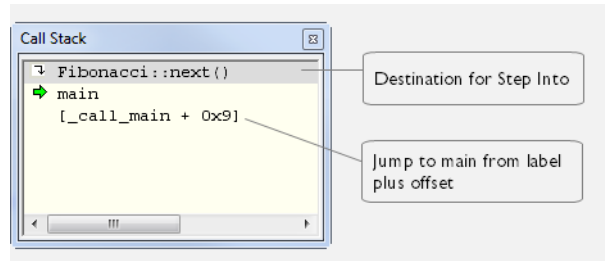
Copies the selected contents of the **Disassembly** window to the clipboard.

Mixed-Mode

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

Call Stack window

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



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

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

Requirements

None; this window is always available.

Display area

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

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

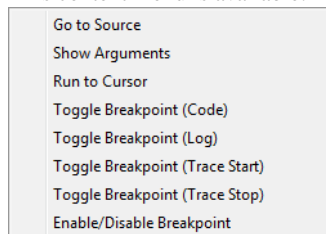
where

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

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

Context menu

This context menu is available:



These commands are available:

Go to Source

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

Show Arguments

Shows function arguments.

Run to Cursor

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

Toggle Breakpoint (Code)

Toggles a code breakpoint.

Toggle Breakpoint (Log)

Toggles a log breakpoint.

Toggle Breakpoint (Trace Start)

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

Toggle Breakpoint (Trace Stop)

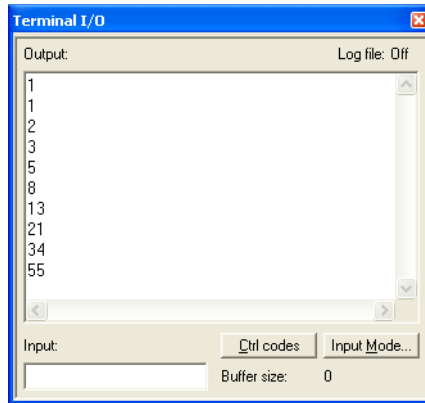
Toggles a Trace Stop breakpoint. When the breakpoint is triggered, trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports it.

Enable/Disable Breakpoint

Enables or disables the selected breakpoint

Terminal I/O window

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



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

To use this window, you must:

- I Link your application with the option **Semihosted** or **IAR breakpoint**.

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.

The following possibilities for using Terminal I/O in real time apply:

Device	Description
Cortex-M	The <code>stdout</code> of your application is routed via SWO. See <i>SWO Configuration dialog box</i> , page 214, specifically the ITM Stimulus Port option.
ARM7/ARM9, including ARMxxx-S, and when using the C-SPY J-Link/J-Trace driver	DCC can be used for Terminal I/O output by adding the file <code>arm\src\debugger\dcc\DCC_Write.c</code> to your project. <code>DCC_write.c</code> overrides the library function <code>write</code> . Functions such as <code>printf</code> can then be used to output text to the Terminal I/O window. In this case, you can disable semihosting which means that the breakpoint it uses is freed for other purposes. To disable semihosting, choose General Options>Library Configuration>Library low-level interface implementation>None .

Table 5: Terminal I/O in real time

Requirements

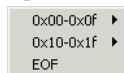
None; this window is always available.

Input

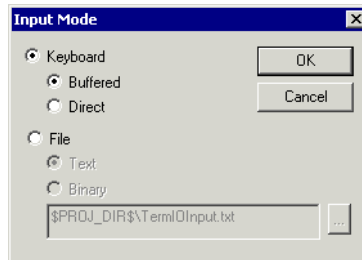
Type the text that you want to input to your application.

Ctrl codes

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

**Input Mode**

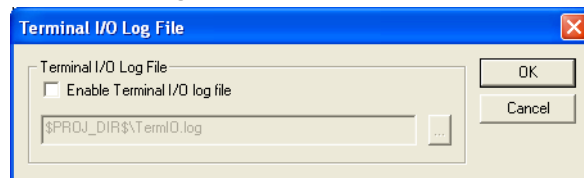
Opens the **Input Mode** dialog box where you choose whether to input data from the keyboard or from a file.



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

Terminal I/O Log File dialog box

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



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

Requirements

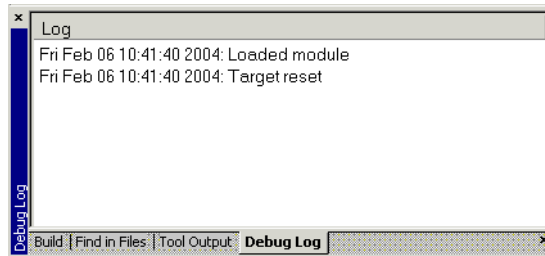
None; this dialog box is always available.

Terminal IO Log Files

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

Debug Log window

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



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

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

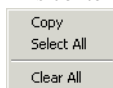
```
<path> (<row>) :<message>
<path> (<row>, <column>) :<message>
```

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

Copy

Copies the contents of the window.

Select All

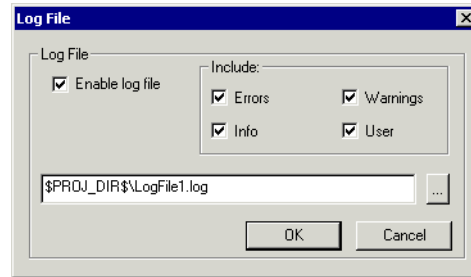
Selects the contents of the window.

Clear All

Clears the contents of the window.

Log File dialog box

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



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

Requirements

None; this dialog box is always available.

Enable Log file

Enables or disables logging to the file.

Include

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

Errors

C-SPY has failed to perform an operation.

Warnings

An error or omission of concern.

Info

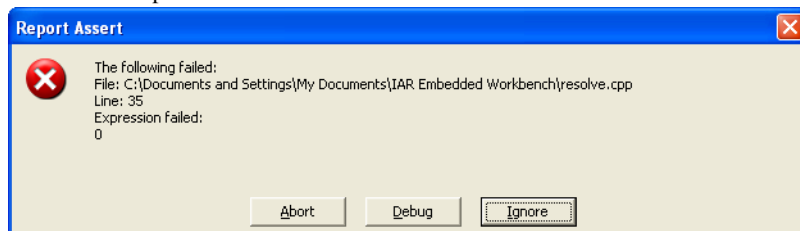
Progress information about actions C-SPY has performed.

User

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

Report Assert dialog box

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



To output the assert message as text:

- 1 Add this function to your application source code:

```
void __aeabi_assert(char const * msg, char const *file, int line)
{
    printf( "%s:%d %s -- assertion failed\n", file, line, msg );
    abort();
}
```

- 2 An assert message is displayed.

Abort

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

Debug

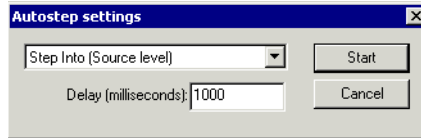
C-SPY stops the execution of the application and returns control to you.

Ignore

The assertion is ignored and the application continues to execute.

Autostep settings dialog box

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



Use this dialog box to customize autostepping.

The drop-down menu lists the available step commands.

Requirements

None; this dialog box is always available.

Delay

Specify the delay between each step in milliseconds.

Variables and expressions

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

Introduction to working with variables and expressions

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

These topics are covered:

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

BRIEFLY ABOUT WORKING WITH VARIABLES AND EXPRESSIONS

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

- **Tooltip watch**—in the editor window—provides the simplest way of viewing the value of a variable or more complex expressions. Just point at the variable with the mouse pointer. The value is displayed next to the variable.
- The **Auto** window displays a useful selection of variables and expressions in, or near, the current statement. The window is automatically updated when execution stops.
- The **Locals** window displays the local variables, that is, auto variables and function parameters for the active function. The window is automatically updated when execution stops.
- The **Watch** window allows you to monitor the values of C-SPY expressions and variables. The window is automatically updated when execution stops.
- The **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.
- 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 **Event Log** window and the **Event Log Summary** window display *event logs* produced when the execution passes specific positions in your application code. The **Timeline** window graphically displays these event logs correlated to a common time-axis. Event logging can help you to analyze program flow and inspect data correlated to a certain position in your application code.

The Cortex ITM communication channels are used for passing events from a running application to the C-SPY Event log system. There are predefined preprocessor macros that you can use in your application source code. An Event log will be generated every time such macros are passed during program execution. You can pass a value with each event. Typically, this value can be either an identifier or the content of a variable or a register (for example, the stack pointer). The value can be written in 8, 16, or 32-bit format. Using a smaller size will reduce the bandwidth needed on the SWO wire. Events can be generated with or without an associated PC (program counter) value, the PC value makes it possible for the debugger to correlate the event to the executed code.

- The Trace-related windows let you inspect the program flow up to a specific state. For more information, see *Trace*, page 199.

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 54.

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.

Table 6: C-SPY assembler symbols expressions

Example	What it does
<code>myVar = #label</code>	Sets <code>myVar</code> to the value of an integer at the address of <code>label</code> .
<code>myPtr = &#label7</code>	Sets <code>myPtr</code> to an <code>int *</code> pointer pointing at <code>label7</code> .

Table 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
<code>#PC</code>	Refers to the program counter.
<code>#`PC`</code>	Refers to the assembler label <code>PC</code> .

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

Which processor-specific symbols are available by default can be seen in the **Register** window, using the `CPU Registers` register group. See *Register window*, page 180.

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 367.

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 372.

Using sizeof

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

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

The former is for types and the latter for expressions.

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

LIMITATIONS ON VARIABLE INFORMATION

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

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

Effects of optimizations

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

Consider this example:

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

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

```
Unavailable
```

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

Working with variables and expressions

These tasks are covered:

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

- Getting started using event logging.

USING THE WINDOWS RELATED TO VARIABLES AND EXPRESSIONS

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

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



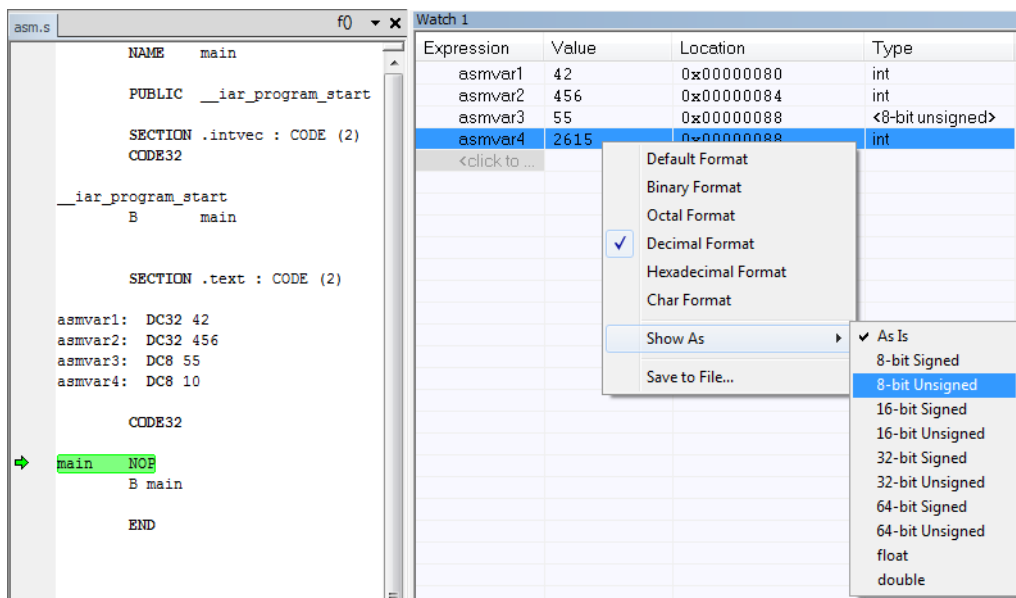
For text that is too wide to fit in a column—in any of these windows, except the **Trace** window—and thus is truncated, just point at the text with the mouse pointer and tooltip information is displayed.

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.

GETTING STARTED USING DATA LOGGING

- 1 To set up for data logging, choose **C-SPY driver>SWO Configuration**. In the dialog box, set up the serial-wire output communication channel for trace data. Note specifically the **CPU clock** option. You can set a default value for the CPU clock on the **Project>Options>C-SPY driver** page. In the **SWO Configuration** dialog box, you can override the default value.

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

- 2 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 about data log breakpoints, see *Data Log breakpoints*, page 127.

- 3 Choose **C-SPY driver>Data Log** to open the **Data Log** window. Optionally, you can also choose:
 - **C-SPY driver>Data Log Summary** to open the **Data Log Summary** window
 - **C-SPY driver>Timeline** to open the **Timeline** window to view the Data Log graph.
- 4 From the context menu, available in the Data Log window, choose **Enable** to enable the logging.
- 5 In the **SWO Configuration** dialog box, you can notice in the Data Log Events area that Data Logs are enabled. Choose which level of logging you want:
 - PC only
 - PC + data value + base addr
 - Data value + exact addr

If you are using the C-SPY simulator you can ignore this step.
- 6 Start executing your application program to collect the log information.
- 7 To view the data log information, look in any of the **Data Log**, **Data Log Summary**, or the Data graph in the **Timeline** window.
- 8 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
- 9 To disable data and interrupt logging, choose **Disable** from the context menu in each window where you have enabled it.

GETTING STARTED USING EVENT LOGGING

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

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

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

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

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

- 3 To change the display format (you can choose between displaying values in hexadecimal or in decimal format), select the event graph for which you want to change the format in the **Timeline** window. Right-click and choose the display format of your choice from the context menu. Note that this setting affects also the **Event Log** window and the **Event Log Summary** window.
- 4 Start executing your application program to collect the log information.

- 5 To view the event information, look at either the **Event Log** window, the **Event Log Summary** window, or the event graph for the specific channel in the **Timeline** window.
- 6 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window.
- 7 To disable event logging, choose **Disable** from the context menu in each window where you have enabled it.

Reference information on working with variables and expressions

Reference information about:

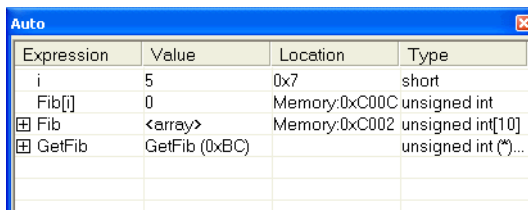
- *Auto window*, page 101
- *Locals window*, page 102
- *Watch window*, page 104
- *Live Watch window*, page 106
- *Statics window*, page 109
- *Quick Watch window*, page 112
- *Symbols window*, page 114
- *Resolve Symbol Ambiguity dialog box*, page 116
- *Data Log window*, page 117
- *Data Log Summary window*, page 119
- *Event Log window*, page 121
- *Event Log Summary window*, page 123

See also:

- *Reference information on trace*, page 207 for trace-related reference information
- *Macro Quicklaunch window*, page 437

Auto window

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



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

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

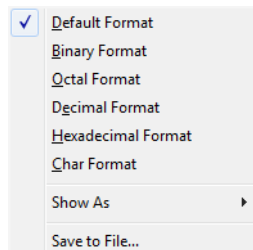
For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 96.

Options

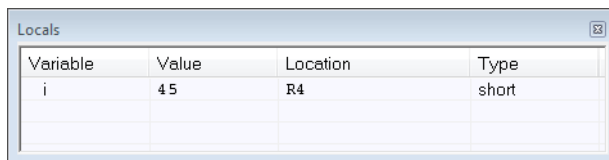
Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context menu in the **Live Watch** window.

Save to File

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

Locals window

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



Variable	Value	Location	Type
i	45	R4	short

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

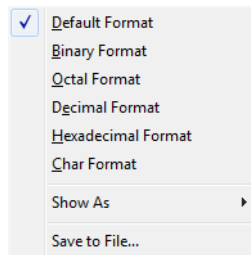
For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 96.

Options

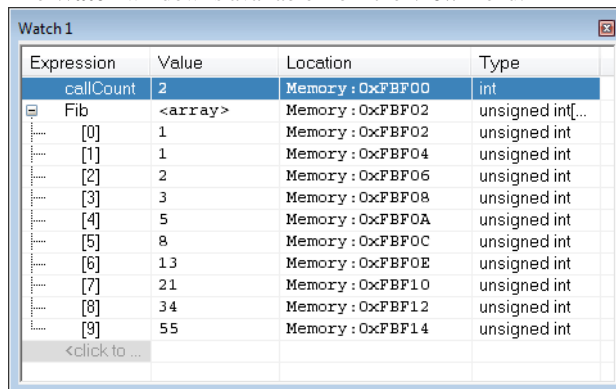
Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context menu in the **Live Watch** window.

Save to File

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

Watch window

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



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

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

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



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

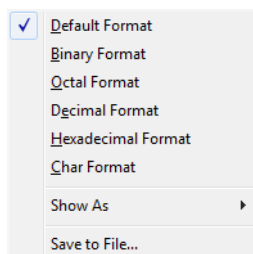
For more information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 96.

Options

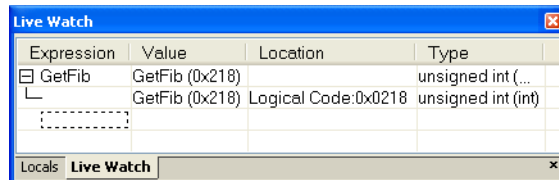
Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context menu in the **Live Watch** window.

Save to File

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

Live Watch window

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



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.

The following possibilities for live watch apply:

Device

Cortex-M	Access to memory or setting breakpoints is always possible during execution.
ARMxxx-S	Setting hardware breakpoints is always possible during execution.

Table 8: Live watch for the different devices

Device

ARM7/ARM9, including ARMxxx-S, and when using the C-SPY J-Link/J-Trace driver	<p>Memory accesses must be made by your application. By adding a small program—a DCC handler—that communicates with the debugger through the DCC unit to your application, memory can be read/written during execution. Software breakpoints can also be set by the DCC handler.</p> <p>Just add the files <code>JLINKDCC_Process.c</code> and <code>JLINKDCC_HandleDataAbort.s</code> located in <code>arm\src\debugger\dcc</code> to your project and call the <code>JLINKDCC_Process</code> function regularly, for example every millisecond.</p> <p>In your local copy of the <code>cstartup</code> file, modify the interrupt vector table so that data aborts will call the <code>JLINKDCC_HandleDataAbort</code> handler. See also <code>-jlink_dcc_timeout</code>, page 473.</p>
---	---

Table 8: Live watch for the different devices

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

Requirements

Any supported hardware debugger system.

Display area

This area contains these columns:

Expression

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

Value

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

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

This column is editable.

Location

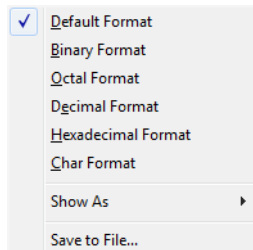
The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

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 96.

Options

Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context menu in the **Live Watch** window.

Save to File

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

Statics window

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

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

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

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

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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

To select variables to monitor:

- 1 In the window, right-click and choose **Select statics** from the context menu. The window now lists all variables with static storage duration.
- 2 Either individually select the variables you want to display, or choose one of the **Select** commands from the context menu.

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

Requirements

None; this window is always available.

Display area

This area contains these columns:

Expression

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

Value

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

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

This column is editable.

Location

The location in memory where this variable is stored.

Type

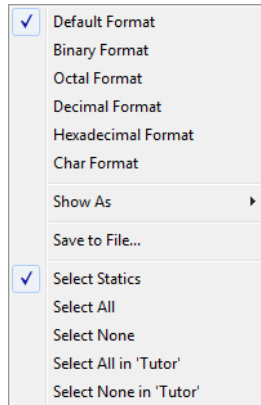
The data type of the variable.

Module

The module of the variable.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Save to File

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

Select Statics

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

Select All

Selects all variables.

Select None

Deselects all variables.

Select All in *module*

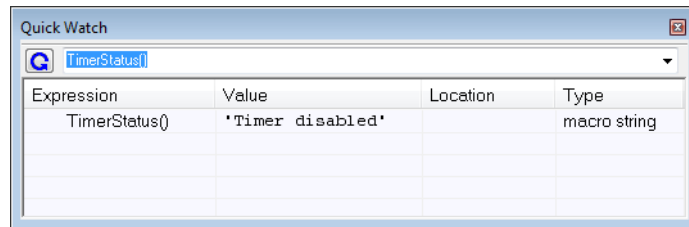
Selects all variables in the selected module.

Select None in *module*

Deselects all variables in the selected module.

Quick Watch window

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



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

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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

To evaluate an expression:

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

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

Alternatively:

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



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

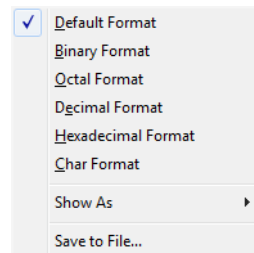
For an example, see *Using C-SPY macros*, page 367.

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 96.

Options

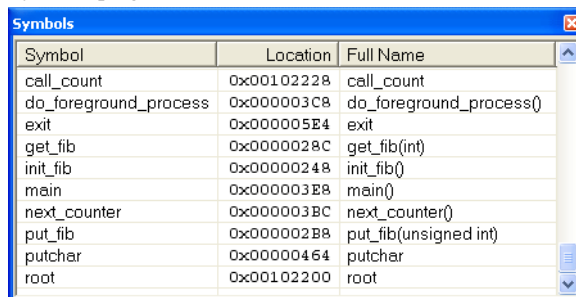
Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context menu in the **Live Watch** window.

Save to File

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

Symbols window

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



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

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

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

Requirements

None; this window is always available.

Display area

This area contains these columns:

Symbol

The symbol name.

Location

The memory address.

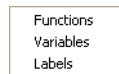
Full name

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

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

Context menu

This context menu is available:



These commands are available:

Functions

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

Variables

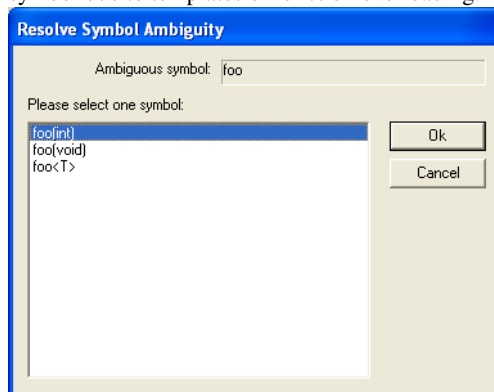
Toggles the display of variables on or off in the list.

Labels

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

Resolve Symbol Ambiguity dialog box

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



Requirements

None; this window is always available.

Ambiguous symbol

Indicates which symbol that is ambiguous.

Please select one symbol

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

Data Log window

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

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

See also *Getting started using data logging*, page 97.

Requirements

A Cortex-M device and one of these alternatives:

- The C-SPY simulator
- The C-SPY I-jet/JTAGjet driver and an I-jet or I-jet Trace in-circuit debugging probe with an SWD interface between the debug probe and the target system
- The C-SPY J-Link/J-Trace driver and a J-Link or J-Trace debug probe with an SWD interface between the debug probe and the target system

For J-Trace, this window is available when ETM trace is disabled. When debugging, this window only displays a limited amount of the collected trace data when ETM is enabled. The entire trace data is displayed when the execution is stopped.

- The C-SPY ST-LINK driver and a ST-LINK debug probe with an SWD interface between the debug probe and the target system

Display area

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

Time

For the I-jet in-circuit debugging probe, the time for the data access is based on a dedicated 48-MHz clock.

The time for the data access for the C-SPY J-Link driver, the C-SPY ST-LINK driver, and the simulator, based on the clock frequency. For the C-SPY J-Link driver and the C-SPY ST-LINK driver, this is specified in the **SWO Configuration** dialog box.

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

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

Cycles

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

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

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

Program Counter*

Displays one of these:

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

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

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

Value

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

To specify what data you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 127.

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 + ?. If you want the offset to be displayed (for the C-SPY I-jet/JTAGjet driver, the C-SPY J-Link driver, and the C-SPY ST-LINK driver), select the **Value + exact addr** option in the **SWO Configuration** dialog box.

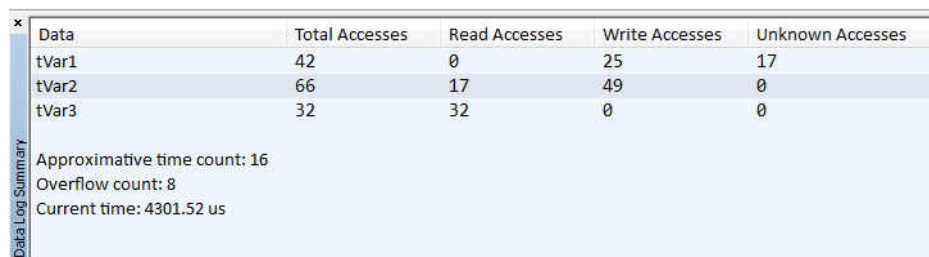
* You can double-click a line in the display area. If the value of the PC for that line is available in the source code, the editor window displays the corresponding source code (this does not include library source code).

Context menu

Identical to the context menu of the **Interrupt Log** window, see *Interrupt Log window*, page 357.

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 97.

Requirements

A Cortex-M device and one of these alternatives:

- The C-SPY simulator
- The C-SPY I-jet/JTAGjet driver and an I-jet or I-jet Trace in-circuit debugging probe with an SWD interface between the debug probe and the target system
- The C-SPY J-Link/J-Trace driver and a J-Link or J-Trace debug probe with an SWD interface between the debug probe and the target system

For J-Trace, this window is available when ETM trace is disabled. When debugging, this window only displays a limited amount of the collected trace data when ETM is enabled. The entire trace data is displayed when the execution is stopped.

- The C-SPY ST-LINK driver and a ST-LINK debug probe with an SWD interface between the debug probe and the target system

Display area

Each row in this area displays the type and the number of accesses to each memory location or area in these columns; and 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 127.

Total Accesses

The number of total accesses.

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

Read Accesses

The number of total read accesses.

Write Accesses

The number of total write accesses.

Unknown Accesses

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

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|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.

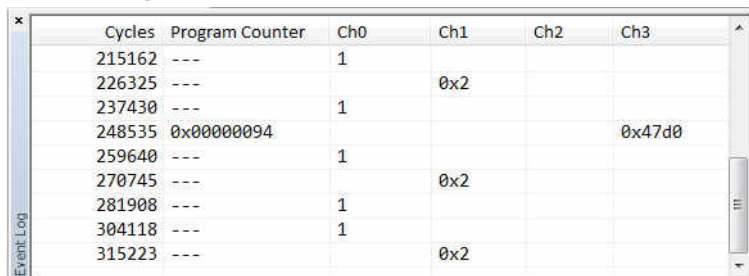
For other C-SPY drivers, the number represents the current time or cycles—the number of cycles or the execution time since the start of execution.

Context menu

Identical to the context menu of the **Interrupt Log** window, see *Interrupt Log window*, page 357.

Event Log window

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



Cycles	Program Counter	Ch0	Ch1	Ch2	Ch3
215162	---	1			
226325	---		0x2		
237430	---	1			
248535	0x00000094				0x47d0
259640	---	1			
270745	---		0x2		
281908	---	1			
304118	---	1			
315223	---		0x2		

This window displays events produced when the execution passes specific positions in your application code. The Cortex ITM communication channels are used for passing the events from a running application to the C-SPY Events system.

See also *Getting started using event logging*, page 99.

Requirements

A Cortex device and one of these alternatives:

- The C-SPY I-jet/JTAGjet driver and an I-jet or I-jet Trace in-circuit debugging probe with an SWD interface between the debug probe and the target system
- The C-SPY J-Link/J-Trace driver and a J-Link or J-Trace debug probe with an SWD interface between the debug probe and the target system

Display area

Each row in the display area shows the events in these columns:

Cycles

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

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

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

Program Counter

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

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

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

ITM1

ITM2

ITM3

ITM4

The Cortex ITM communication channels for which the events are logged. For each event, the event value is displayed.

Add a preprocessor macro to your application source code where you want events to be generated. See *Getting started using event logging*, page 99.

Context menu

Identical to the context menu of the **Interrupt Log** window, see *Interrupt Log window*, page 357.

Event Log Summary window

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

Channel	Count	Average Value	Min Value	Max Value	Average Interval	Min Interval	Max Interval
Ch0	13	1	1	1	506.720us	444.200us	1189.980us
Ch1	7	0x2	0x2	0x2	889.360us	888.400us	889.620us
Ch2	0						
Ch3	1	0x47d0	0x47d0	0x47d0			

Approximative time count: 0
 Overflow count: 0
 Current time: 6507.580us

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

Requirements

A Cortex device and one of these alternatives:

- The C-SPY I-jet/JTAGjet driver and an I-jet or I-jet Trace in-circuit debugging probe with an SWD interface between the debug probe and the target system
- The C-SPY J-Link/J-Trace driver and a J-Link or J-Trace debug probe with an SWD interface between the debug probe and the target system

Display area

Each row displays the type and the number of accesses to each location in your application code in these columns; and summary information is listed at the bottom of the display area:

Channel

The name of the communication channel for which events are generated.

Count

The number of logged events.

Average Value

The average value of all received event values.

Min Value

The smallest value of all received event values.

Max Value

The largest value of all received event values.

Average Interval

The average time (in cycles) between events.

Min Interval

The shortest time (in cycles) between two events.

Max Interval

The longest time (in cycles) between two events.

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|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

Identical to the context menu of the **Interrupt Log** window, see *Interrupt Log window*, page 357.

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
- *Breakpoints options*, page 130

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 129.

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.

Log breakpoints

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

Trace Start and Stop breakpoints

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

Data breakpoints

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

Data Log breakpoints

Data log breakpoints are triggered when a specified variable is accessed. A log entry is written in the **SWO Trace** window (**Trace** window in the simulator) for each access. A log message can also be displayed in the **Data Log** window. Data logs can also be displayed on the Data Log graph in the **Timeline** window, if that window is enabled. However, these log messages require that you have set up trace data in the **SWO Configuration** dialog box, see *SWO Configuration dialog box*, page 214.

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

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

Immediate breakpoints

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

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

JTAG watchpoints

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

The watchpoints are implemented using the functionality provided by the ARM EmbeddedICE™ macrocell. The macrocell is part of every ARM core that supports the JTAG interface. The EmbeddedICE watchpoint comparator compares the address bus,

data bus, CPU control signals and external input signals with the defined watchpoint in real time. When all defined conditions are true, the program will break.

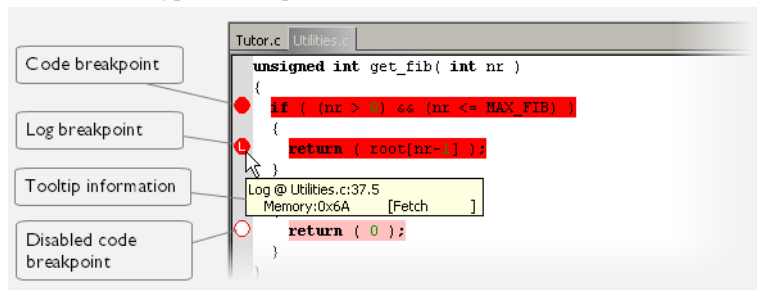
The watchpoints are implicitly used by C-SPY to set code breakpoints or data breakpoints in the application. When setting breakpoints in read/write memory, only one watchpoint is needed by the debugger. When setting breakpoints in read-only memory, one watchpoint is needed for each breakpoint. Because the macrocell only implements two hardware watchpoints, the maximum number of breakpoints in read-only memory is two.

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

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

BREAKPOINT ICONS

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



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



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

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

BREAKPOINTS IN THE C-SPY SIMULATOR

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

BREAKPOINTS IN THE C-SPY HARDWARE DEBUGGER DRIVERS

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

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

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

BREAKPOINT CONSUMERS

A debugger system includes several consumers of breakpoints.

User breakpoints

The breakpoints you define in the breakpoint dialog box or by toggling breakpoints in the editor window often consume one physical breakpoint each, but this can vary greatly. Some user breakpoints consume several physical breakpoints and conversely, several user breakpoints can share one physical breakpoint. User breakpoints are displayed in the same way both in the **Breakpoint Usage** 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 **Semihosted** or **IAR breakpoint** 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.

BREAKPOINTS OPTIONS

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

- GDB Server
- I-jet/JTAGjet
- J-Link/J-Trace
- CMSIS-DAP
- Macraigor.

For more information, see *Breakpoints options dialog box*, page 153.

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
- Setting a breakpoint on an exception vector
- Setting breakpoints in `__ramfunc` declared functions
- Useful breakpoint hints.

VARIOUS WAYS TO SET A BREAKPOINT

You can set a breakpoint in various ways:

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

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

TOGGLING A SIMPLE CODE BREAKPOINT

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



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

SETTING BREAKPOINTS USING THE DIALOG BOX

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

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

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

To set a new breakpoint:

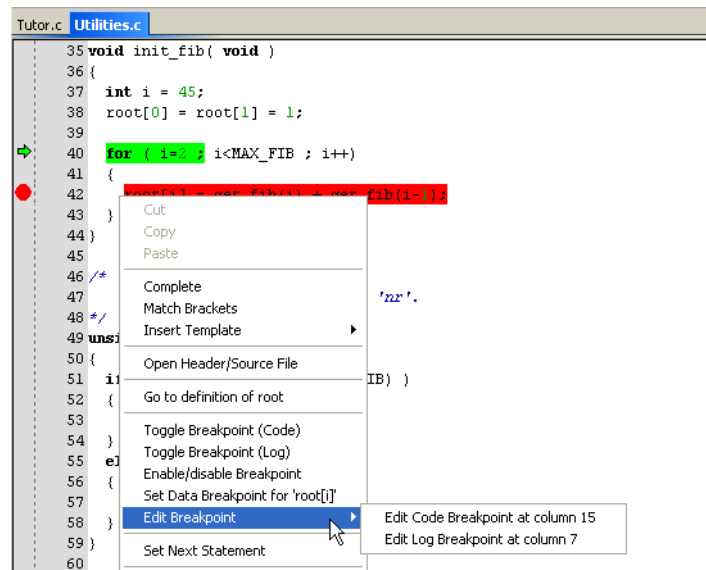
- 1 Choose **View>Breakpoints** to open the **Breakpoints** window.
- 2 In the **Breakpoints** window, right-click, and choose **New Breakpoint** from the context menu.
- 3 On the submenu, choose the breakpoint type you want to set.

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

- 4 In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.
The breakpoint is displayed in the **Breakpoints** window.

To modify an existing breakpoint:

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



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

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

SETTING A DATA BREAKPOINT IN THE MEMORY WINDOW

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

The breakpoint is not highlighted in the **Memory** window; instead, you can see, edit, and remove it using the **Breakpoints** window, which is available from the **View** menu. The breakpoints you set in the **Memory** window will be triggered for both read and

write accesses. All breakpoints defined in this window are preserved between debug sessions.

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

SETTING BREAKPOINTS USING SYSTEM MACROS

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

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

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

These breakpoint macros are available:

C-SPY macro for breakpoints	Simulator	I-jet/JTAGjet	J-Link/J-Trace	CMSIS-DAP
__setCodeBreak	Yes	Yes	Yes	Yes
__setDataBreak	Yes	Yes	No	Yes
__setLogBreak	Yes	Yes	Yes	Yes
__setDataLogBreak	Yes	Yes	No	No
__setSimBreak	Yes	No	No	No
__setTraceStartBreak	Yes	Yes	No	No
__setTraceStopBreak	Yes	Yes	No	No
__clearBreak	Yes	Yes	Yes	Yes

Table 9: C-SPY macros for breakpoints

C-SPY macro for breakpoints	GDB Server / Macraigor / RDI	ST-LINK	PE micro	TI Stellaris / TI XDS	Angel / IAR ROM-monitor
__setCodeBreak	Yes	Yes	Yes	Yes	Yes
__setDataBreak	No	No	No	No	No

Table 10: C-SPY macros for breakpoints

C-SPY macro for breakpoints	GDB Server /		PE micro	TI Stellaris / TI XDS	Angel / IAR ROM-monitor
	Macraigor / RDI	ST-LINK			
__setLogBreak	Yes	Yes	Yes	Yes	Yes
__setDataLogBreak	No	No	No	No	No
__setSimBreak	No	No	No	No	No
__setTraceStartBreak	No	No	No	No	No
__setTraceStopBreak	No	No	No	No	No
__clearBreak	Yes	Yes	Yes	Yes	Yes

Table 10: C-SPY macros for breakpoints (Continued)

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 380.

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 367.

SETTING A BREAKPOINT ON AN EXCEPTION VECTOR

You can set breakpoints on exception vectors for ARM9, Cortex-R4, and Cortex-M3 devices. Use the **Vector Catch** dialog box to set a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. For more information, see *Vector Catch dialog box*, page 156.

For the C-SPY I-jet/JTAGjet driver, the C-SPY J-Link/J-Trace driver, and for C-SPY RDI drivers, it is also possible to set breakpoints directly on a vector already in the options dialog box, see *Setup options for J-Link/J-Trace*, page 523 and *RDI*, page 532.

This procedure applies to the C-SPY I-jet/JTAGjet driver, the C-SPY J-Link/J-Trace driver and the C-SPY Macraigor driver.

To set a breakpoint on an exception vector:

- 1 Select the correct device. Before starting C-SPY, choose **Project>Options** and select the **General Options** category. Choose the appropriate core or device from one of the **Processor variant** drop-down lists available on the **Target** page.
- 2 Start C-SPY.
- 3 Choose *C-SPY driver*>**Vector Catch**. By default, vectors are selected according to your settings on the Breakpoints options page, see *Breakpoints options dialog box*, page 153.

- 4 In the **Vector Catch** dialog box, select the vector you want to set a breakpoint on, and click **OK**. The breakpoint will only be triggered at the beginning of the exception.

SETTING BREAKPOINTS IN __RAMFUNC DECLARED FUNCTIONS

To set a breakpoint in a `__ramfunc` declared function, the program execution must have reached the `main` function. The system startup code moves all `__ramfunc` declared functions from their stored location—normally flash memory—to their RAM location, which means the `__ramfunc` declared functions are not in their proper place and breakpoints cannot be set until you have executed up to the `main` function. Use the **Restore software breakpoints** option to solve this problem, see *Breakpoints options dialog box*, page 153, specifically the **Restore software breakpoints** option.

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

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

USEFUL BREAKPOINT HINTS

Below are some useful hints related to setting breakpoints.



Tracing incorrect function arguments

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

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

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

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

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

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

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



Performing a task and continuing execution

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

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

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

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

```
__var my_counter;

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

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

Reference information on breakpoints

Reference information about:

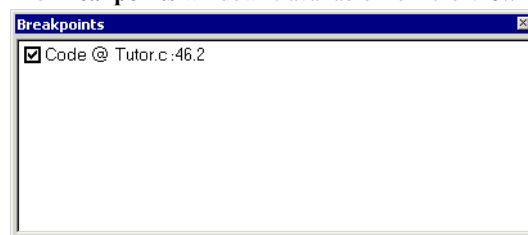
- *Breakpoints window*, page 137
- *Breakpoint Usage window*, page 139
- *Code breakpoints dialog box*, page 140
- *JTAG Watchpoints dialog box*, page 142
- *Log breakpoints dialog box*, page 145
- *Data breakpoints dialog box*, page 146
- *Data Log breakpoints dialog box (C-SPY hardware drivers)*, page 151
- *Data Log breakpoints dialog box (C-SPY hardware drivers)*, page 151
- *Breakpoints options dialog box*, page 153
- *Immediate breakpoints dialog box*, page 155
- *Vector Catch dialog box*, page 156
- *Enter Location dialog box*, page 157
- *Resolve Source Ambiguity dialog box*, page 158.

See also:

- *Reference information on C-SPY system macros*, page 380
- *Reference information on trace*, page 207.

Breakpoints window

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



This window lists all breakpoints you define.

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

Requirements

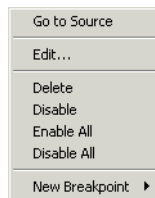
None; this window is always available.

Display area

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

Context menu

This context menu is available:



These commands are available:

Go to Source

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

Edit

Opens the breakpoint dialog box for the breakpoint you selected.

Delete

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

Enable

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

Disable

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

Enable All

Enables all defined breakpoints.

Disable All

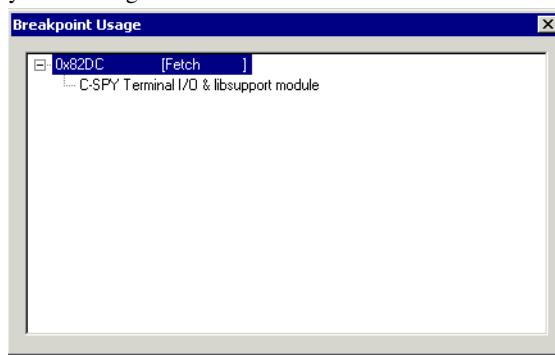
Disables all defined breakpoints.

New Breakpoint

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

Breakpoint Usage window

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



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

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

C-SPY uses breakpoints when stepping. 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 129.

Requirements

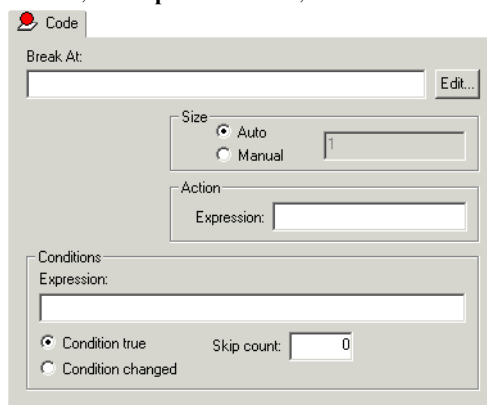
None; this window is always available.

Display area

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

Code breakpoints dialog box

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



This figure reflects the C-SPY simulator.

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

Requirements

None; this dialog box is always available.

Break At

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 157.

Breakpoint type

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

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

- The C-SPY I-jet/JTAGjet driver
- The C-SPY CMSIS-DAP driver
- The C-SPY GDB Server driver

- The C-SPY J-Link/J-Trace driver
- The C-SPY Macraigor JTAG driver.

Size

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

Auto

The size will be set automatically, typically to 1.

Manual

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

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 135.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 92.

Condition true

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

Condition changed

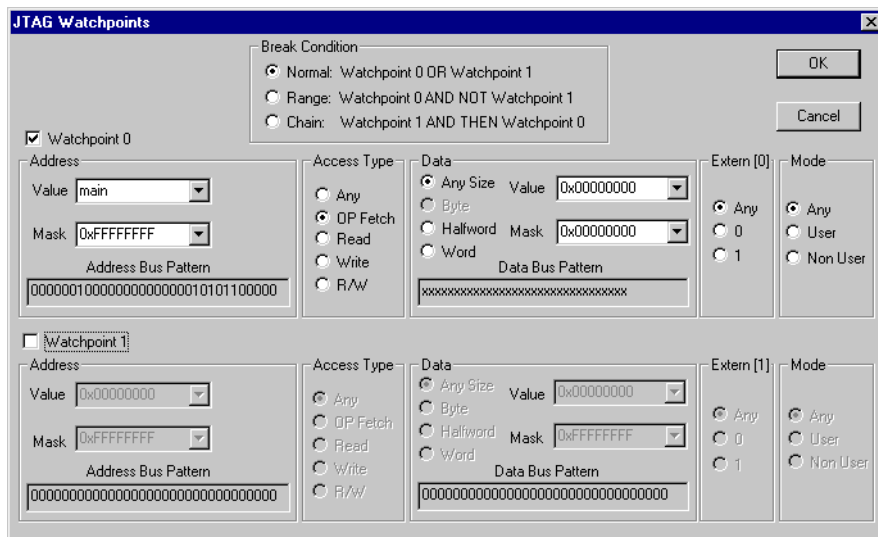
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Skip count

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

JTAG Watchpoints dialog box

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



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

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

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

Requirements

One of these alternatives:

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

Address

Specify the address to watch for.

Value	Specify an address or a C-SPY expression that evaluates to an address. Alternatively, you can select an address you have previously watched for from the drop-down list. For detailed information about C-SPY expressions, see <i>C-SPY expressions</i> , page 92.
Mask	Qualifies each bit in the value. A zero bit in the mask will cause the corresponding bit in the value to be ignored in the comparison. To match any address, enter 0. Note that the mask values are inverted with respect to the notation used in the ARM hardware manuals.
Address Bus Pattern	Shows the bit pattern to be used by the address comparator. Ignored bits as specified in the mask are shown as x.

Access Type

Selects the access type of the data to watch for:

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

Data

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

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

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

Value	Specify a value or a C-SPY expression. Alternatively, you can select a value you have previously watched for from the drop-down list. For detailed information about C-SPY expressions, see <i>C-SPY expressions</i> , page 92.
Mask	Qualifies each bit in the value. A zero bit in the mask will cause the corresponding bit in the value to be ignored in the comparison. To match any address, enter 0. Note that the mask values are inverted with respect to the notation used in the ARM hardware manuals.
Data Bus Pattern	Shows the bit pattern to be used by the address comparator. Ignored bits as specified in the mask are shown as x.

Extern

Defines the state of the external input. Choose between:

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

Mode

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

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

Break Condition

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

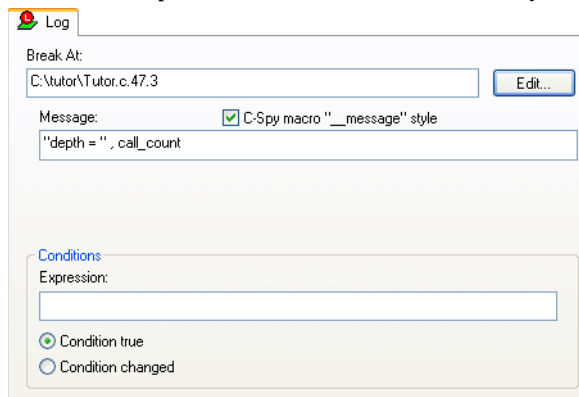
Normal	Uses the two watchpoints individually (OR).
Range	Combines both watchpoints to cover a range where watchpoint 0 defines the start of the range and watchpoint 1 the end of the range. Selectable ranges are restricted to being powers of 2.

Chain

Makes a trigger of watchpoint 1 and watchpoint 0. A program break will then occur when watchpoint 0 is triggered.

Log breakpoints dialog box

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



This figure reflects the C-SPY simulator.

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

Requirements

None; this dialog box is always available.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 157.

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 375.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 92.

Condition true

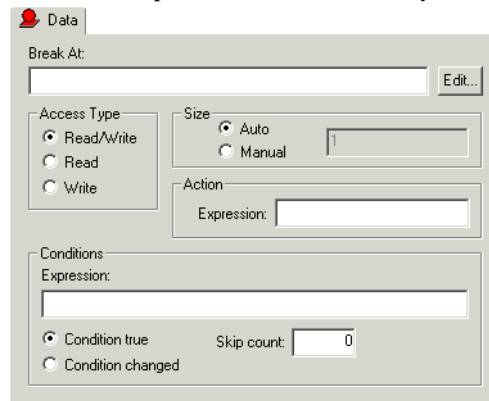
The breakpoint is triggered if the value of the expression is true.

Condition changed

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

Data breakpoints dialog box

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



This figure reflects the C-SPY simulator.

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

Requirements

One of these alternatives:

- The C-SPY simulator
- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver
- The C-SPY RDI driver
- The C-SPY CMSIS-DAP driver
- The C-SPY Macraigor driver
- The C-SPY GDB Server driver
- The C-SPY TI Stellaris driver

Break At

Specify the data location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 157.

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 135.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 92.

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.

Trigger range

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

Extend to cover requested range

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

The **Trigger range** option is available for all C-SPY hardware drivers that support data breakpoints.

Match data

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

Value

Specify a data value.

Mask

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

For Cortex-M, the data mask is limited to one of these exact values:

0xFFFFFFFF, which means that the complete word must match.

0xFFFF, which means that the match can be either the upper or lower 16-bit part of a word or halfword.

0xFF, which means that the match can be either the upper, middle, or lower 8-bit part of a word, halfword, or byte. For example, for the data 0xVV, any 32-bit access matching a xxxxxxVV, xxxxVVxx, xxVVxxxx, or VVxxxxxx pattern, and any 16-bit access matching xxVV or VVxx, and an 8-bit access with exact match triggers the breakpoint.

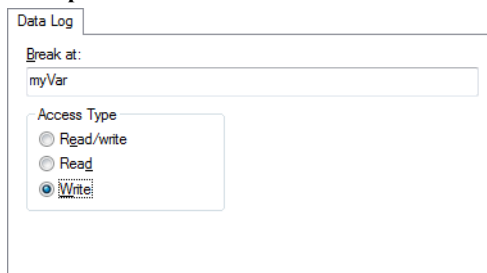
The **Match data** options are only available for I-jet, I-jet Trace, JTAGjet, J-Link/J-Trace and ST-LINK, and when using an ARM7/9 or a Cortex-M device.

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

Note: The **Match Data** options are not available for Cortex-M0, Cortex-M1, and Cortex-M0+.

Data Log breakpoints dialog box

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



This figure reflects the C-SPY simulator.

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

See also *Data Log breakpoints*, page 127 and *Getting started using data logging*, page 97.

Requirements

The C-SPY simulator.

Break At

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

Access Type

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

Read/Write

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

Read

Read accesses from the location of the variable.

Write

Write accesses to location of the variable.

Data Log breakpoints dialog box (C-SPY hardware drivers)

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

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

You can set a data log breakpoint on 8-, 16-, and 32-bit variables.

See also *Data Log breakpoints*, page 127 and *Getting started using data logging*, page 97.

Requirements

One of these alternatives:

- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver.

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 157.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read/Write

Reads from or writes to location.

Read

Reads from location; except for Cortex-M3, revision 1 devices.

Write

Writes to location; except for Cortex-M3, revision 1 devices.

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.

Trigger range

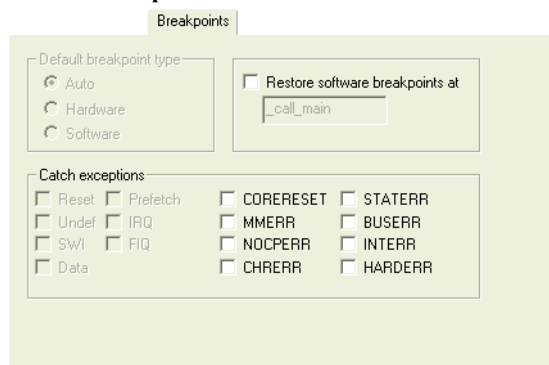
Shows the requested range and the effective range to be covered by the trace. The range suggested is either within or exactly the area specified by the **Trigger at** and the **Size** options.

Extend to cover requested range

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

Breakpoints options dialog box

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



Use this dialog box to set driver-specific breakpoint options.

Requirements

One of these alternatives:

- The C-SPY CMSIS-DAP driver
- The C-SPY GDB Server driver
- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY Macraigor driver
- The C-SPY ST-LINK driver
- The C-SPY TI XDS driver

Default breakpoint type

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

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

Restore software breakpoints at

Automatically restores any breakpoints that were destroyed during system startup.

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

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

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

Catch exceptions

Sets a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. This option is available for ARM9, Cortex-R4, and Cortex-M3 devices. The settings you make will work as default settings for the project. However, you can override these default settings during the debug session by using the **Vector Catch** dialog box, see *Setting a breakpoint on an exception vector*, page 134.

The settings you make will be preserved during debug sessions.

This option is supported by the C-SPY I-jet/JTAGjet driver and the C-SPY J-Link/J-Trace driver.

Immediate breakpoints dialog box

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

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

Requirements

The C-SPY simulator.

Trigger at

Specify the data location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 157.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read

Reads from location.

Write

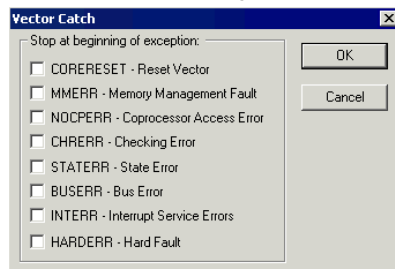
Writes to location.

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 135.

Vector Catch dialog box

The **Vector Catch** dialog box is available from the C-SPY driver menu.



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

This figure reflects a Cortex-M device. If you are using another device, the contents of this dialog box might look different.

Note: For the C-SPY I-jet/JTAGjet driver, the C-SPY J-Link/J-Trace driver, and for C-SPY RDI drivers, it is also possible to set breakpoints directly on a vector already in the options dialog box, see *Setup options for J-Link/J-Trace*, page 523 and *RDI*, page 532.

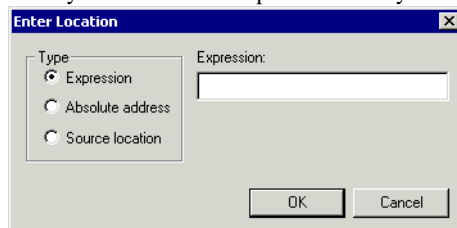
Requirements

One of these alternatives:

- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY Macraigor driver

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 92.

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 163.

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.

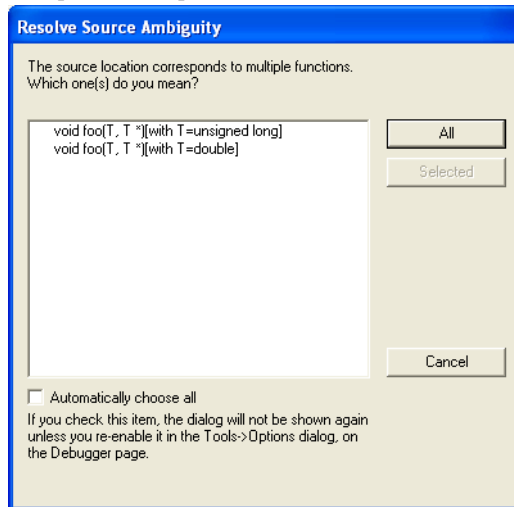
For example, `{C:\src\prog.c}.22.3`

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 ARM*.

Memory and registers

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

Introduction to monitoring memory and registers

These topics are covered:

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

BRIEFLY ABOUT MONITORING MEMORY AND REGISTERS

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

- The **Memory** window

Gives an up-to-date display of a specified area of memory—a memory zone—and allows you to edit it. Different colors are used for indicating data coverage along with execution of your application. You can fill specified areas with specific values and you can set breakpoints directly on a memory location or range. You can open several instances of this window, to monitor different memory areas. The content of the window can be regularly updated while your application is executing.
- The **Symbolic Memory** window

Displays how variables with static storage duration are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example by buffer overruns.
- The **Stack** window

Displays the contents of the stack, including how stack variables are laid out in memory. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the **Stack** window is useful for determining the optimal size of the stack. You can open up to two instances of this window, each showing different stacks or different display modes of the same stack.

- The **Register** window

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

- The **SFR Setup** window

Displays the currently defined SFRs that C-SPY has information about. If required, you can use this window to customize aspects of the SFRs.

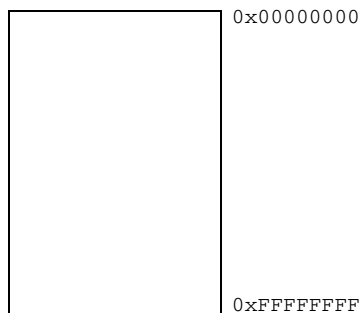
To view the memory contents for a specific variable, simply drag the variable to the **Memory** window or the **Symbolic** memory window. The memory area where the variable is located will appear.



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

C-SPY MEMORY ZONES

In C-SPY, the term *zone* is used for a named memory area. A memory address, or *location*, is a combination of a zone and a numerical offset into that zone. The ARM architecture has only one zone, `Memory`, which covers the whole ARM memory range.



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.

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

Note: For the C-SPY I-jet/JTAGjet driver, you can specify the automatic selection of access width in the **Edit Memory Range** dialog box; see *Edit Memory Range dialog box, for C-SPY hardware debugger drivers*, page 194.

STACK DISPLAY

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

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

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

Stack usage

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



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

Note: The size and location of the stack is retrieved from the definition of the section holding the stack, made in the linker configuration file. If you, for some reason, modify the stack initialization made in the system startup code, `cstartup`, you should also change the section definition in the linker configuration file accordingly; otherwise the Stack window cannot track the stack usage. For more information about this, see the *IAR C/C++ Development Guide for ARM*.

MEMORY ACCESS CHECKING

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

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

Monitoring memory and registers

These tasks are covered:

- *Configuring C-SPY to match the memory of your device*, page 165.
- *Defining application-specific register groups*, page 165.

CONFIGURING C-SPY TO MATCH THE MEMORY OF YOUR DEVICE

Typically, when you set up your project, a device description file for your particular device is automatically or manually selected. If that file fully specifies the memory range information for your device, you do not have to configure C-SPY in this respect.

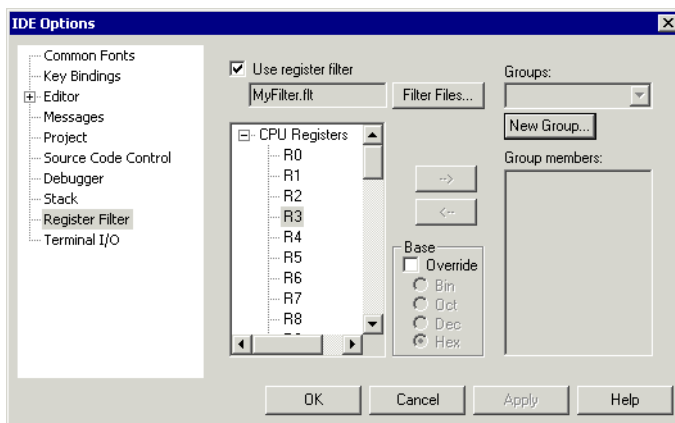
However, if that file does not specify memory ranges for a specific device, but instead for a family of devices (perhaps with varying amounts of on-chip RAM), you should tailor the areas to match your device.

To fine-tune the memory areas to suit your device, use the *Memory Configuration dialog box, for the C-SPY simulator*, page 187 and *Memory Configuration dialog box, in C-SPY hardware debugger drivers*, page 191.

DEFINING APPLICATION-SPECIFIC REGISTER GROUPS

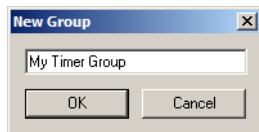
Defining application-specific register groups minimizes the amount of registers displayed in the **Register** window and speeds up the debugging.

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



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

- 2 Select **Use register filter** and specify the filename and destination of the filter file for your new group in the dialog box that appears.
- 3 Click **New Group** and specify the name of your group, for example **My Timer Group**.



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

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

Note: The registers that appear in the list of registers are retrieved from the `ddf` file that is currently used. If a certain SFR that you need does not appear, you can register your own SFRs. For more information, see *SFR Setup window*, page 183.

Reference information on memory and registers

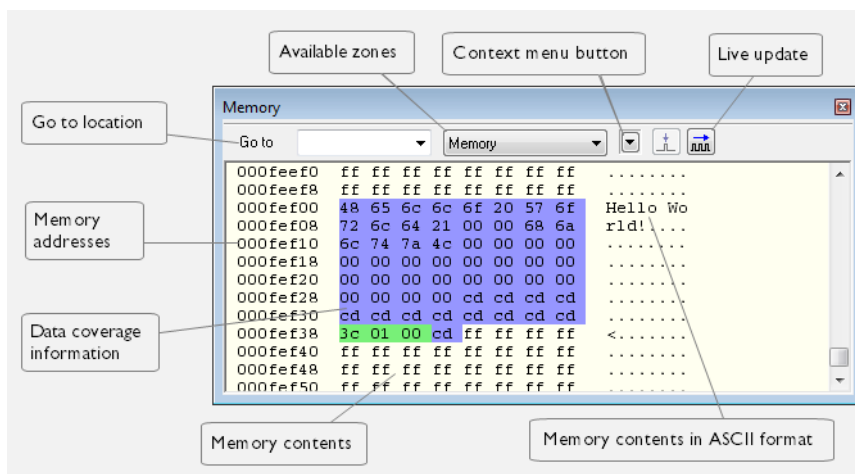
Reference information about:

- *Memory window*, page 167
- *Memory Save dialog box*, page 171
- *Memory Restore dialog box*, page 172
- *Fill dialog box*, page 173
- *Symbolic Memory window*, page 174
- *Stack window*, page 177
- *Register window*, page 180
- *SFR Setup window*, page 183
- *Edit SFR dialog box*, page 186

- *Memory Configuration dialog box, for the C-SPY simulator, page 187*
- *Edit Memory Range dialog box, for the C-SPY simulator, page 190*
- *Memory Configuration dialog box, in C-SPY hardware debugger drivers, page 191*
- *Edit Memory Range dialog box, for C-SPY hardware debugger drivers, page 194*

Memory window

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



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



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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Context menu button

Displays the context menu.

Update Now

Updates the content of the **Memory** window while your application is executing. This button is only enabled if the C-SPY driver you are using has access to the target system memory while your application is executing.

Live Update

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

Display area

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

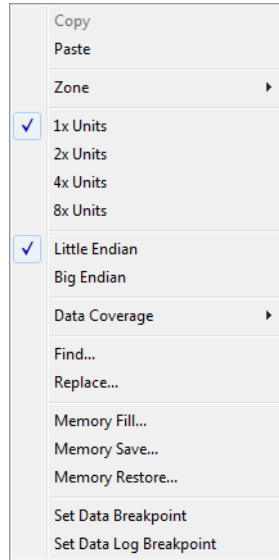
Data coverage is displayed with these colors:

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

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

Context menu

This context menu is available:



These commands are available:

Copy, Paste

Standard editing commands.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

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 ARM*.

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 ARM*.

Memory Fill

Displays a dialog box, where you can fill a specified area with a value, see *Fill dialog box*, page 173.

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 171.

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 172.

Set Data Breakpoint

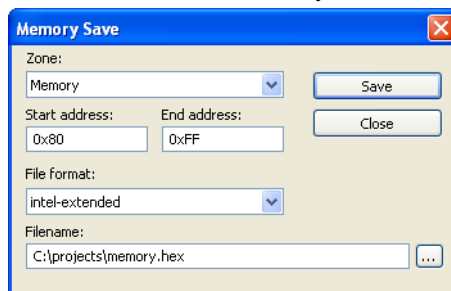
Sets breakpoints directly in the **Memory** window. The breakpoint is not highlighted; you can see, edit, and remove it in the **Breakpoints** dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the Memory window*, page 132.

Set Data Log Breakpoint

Sets a breakpoint on the start address of a memory selection directly in the **Memory** window. The breakpoint is not highlighted; you can see, edit, and remove it in the **Breakpoints** dialog box. The breakpoints you set in this window will be triggered by both read and write accesses; to change this, use the **Breakpoints** window. For more information, see *Data Log breakpoints*, page 127 and *Getting started using data logging*, page 97.

Memory Save dialog box

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



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

Requirements

None; this dialog box is always available.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Start address

Specify the start address of the memory range to be saved.

End address

Specify the end address of the memory range to be saved.

File format

Selects the file format to be used, which is Intel-extended by default.

Filename

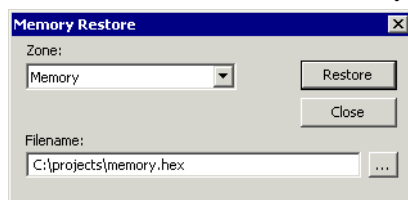
Specify the destination file to be used; a browse button is available for your convenience.

Save

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

Memory Restore dialog box

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



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

Requirements

None; this dialog box is always available.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Filename

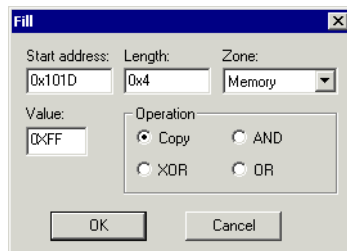
Specify the file to be read; a browse button is available for your convenience.

Restore

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

Fill dialog box

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



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

Requirements

None; this dialog box is always available.

Start address

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

Length

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

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Value

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

Operation

These are the available memory fill operations:

Copy

Value will be copied to the specified memory area.

AND

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

XOR

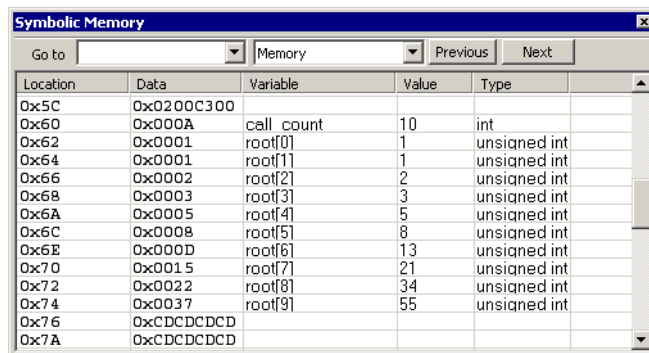
An XOR operation will be performed between Value and the existing contents of memory before writing the result to memory.

OR

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

Symbolic Memory window

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



Location	Data	Variable	Value	Type
0x5C	0x0200C300			
0x60	0x000A	call count	10	int
0x62	0x0001	root[0]	1	unsigned int
0x64	0x0001	root[1]	1	unsigned int
0x66	0x0002	root[2]	2	unsigned int
0x68	0x0003	root[3]	3	unsigned int
0x6A	0x0005	root[4]	5	unsigned int
0x6C	0x0008	root[5]	8	unsigned int
0x6E	0x000D	root[6]	13	unsigned int
0x70	0x0015	root[7]	21	unsigned int
0x72	0x0022	root[8]	34	unsigned int
0x74	0x0037	root[9]	55	unsigned int
0x76	0xCDCDCDCD			
0x7A	0xCDCDCDCD			

This window displays how variables with static storage duration, typically variables with file scope but also static variables in functions and classes, are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example buffer overruns. Other areas of use are spotting alignment holes or for understanding problems caused by buffers being overwritten.



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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Previous

Highlights the previous symbol in the display area.

Next

Highlights the next symbol in the display area.

Display area

This area contains these columns:

Location

The memory address.

Data

The memory contents in hexadecimal format. The data is grouped according to the size of the symbol. This column is editable.

Variable

The variable name; requires that the variable has a fixed memory location. Local variables are not displayed.

Value

The value of the variable. This column is editable.

Type

The type of the variable.

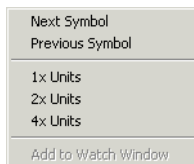
There are several different ways to navigate within the memory space:

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

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

Context menu

This context menu is available:



These commands are available:

Next Symbol

Highlights the next symbol in the display area.

Previous Symbol

Highlights the previous symbol in the display area.

1x Units

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

2x Units

Displays the memory contents as 2-byte groups.

4x Units

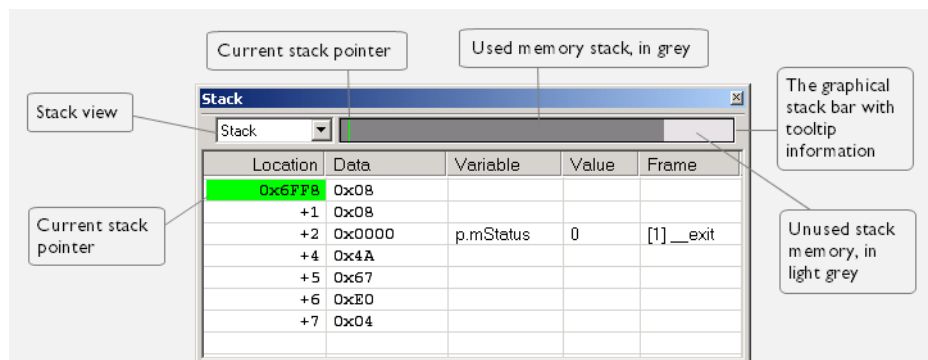
Displays the memory contents as 4-byte groups.

Add to Watch window

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

Stack window

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



This window is a memory window that displays the contents of the stack. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the **Stack** window is useful for determining the optimal size of the stack.

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

For applications that set up the stacks using other mechanisms, it is possible to override the default mechanism. Use one of the C-SPY command line option variants, see `--proc_stack_stack`, page 482.

To view the graphical stack bar:

- 1 Choose **Tools>Options>Stack**.
- 2 Select the option **Enable graphical stack display and stack usage**.

You can open up to two **Stack** windows, each showing a different stack—if several stacks are available—or the same stack with different display settings.

Note: By default, this window uses one physical breakpoint. For more information, see *Breakpoint consumers*, page 129.

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

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Stack

Selects which stack to view. This applies to cores with multiple stacks.

The graphical stack bar

Displays the state of the stack graphically.

The left end of the stack bar represents the bottom of the stack, in other words, the position of the stack pointer when the stack is empty. The right end represents the end of the memory space reserved for the stack. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

When the stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled.



Place the mouse pointer over the stack bar to get tooltip information about stack usage.

Display area

This area contains these columns:

Location

Displays the location in memory. The addresses are displayed in increasing order. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.

Data

Displays the contents of the memory unit at the given location. From the **Stack** window context menu, you can select how the data should be displayed; as a 1-, 2-, or 4-byte group of data.

Variable

Displays the name of a variable, if there is a local variable at the given location. Variables are only displayed if they are declared locally in a function, and located on the stack and not in registers.

Value

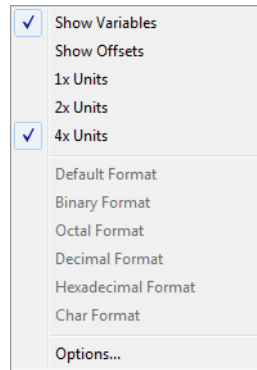
Displays the value of the variable that is displayed in the **Variable** column.

Frame

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

Context menu

This context menu is available:



These commands are available:

Show variables

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

Show offsets

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

1x Units

Displays the memory contents as single bytes.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

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

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

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

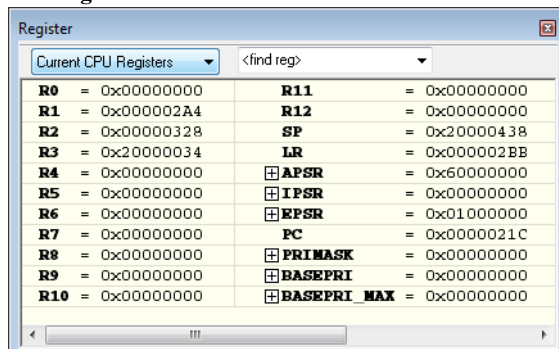
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration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

Options

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

Register window

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



This window gives an up-to-date display of the contents of the processor registers and special function registers, and allows you to edit the content of some of the registers.

Optionally, you can choose to load either predefined register groups or to define your own application-specific groups.

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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 61.

To enable predefined register groups:

- 1 Select a device description file that suits your device, see *Selecting a device description file*, page 51.
- 2 The register groups appear in the **Register** window, provided that they are defined in the device description file. Note that the available register groups are also listed on the **Register Filter** page.

To define application-specific register groups:

See *Defining application-specific register groups*, page 165.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

CPU Registers

Selects which register group to display, by default **CPU Registers**. By default, there are two register groups in the debugger: If some of your SFRs are missing, you can register your own SFRs in a Custom group, see *SFR Setup window*, page 183.

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

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

Additional register groups are predefined in the device description files—available in the `arm\config` directory—that make all SFR registers available in the register window. The device description file contains a section that defines the special function registers and their groups.

<find register>

Specify the name of a register that you want to find. Press the Enter key and the first group where this register is found is displayed. The register search box has a history depth of 20 search entries.

Display area

Displays registers and their values. Every time C-SPY stops, a value that has changed since the last stop is highlighted. Some of the registers are read-only, some of the registers are write-only (marked with *w*), and some of the registers are editable. To edit the contents of an editable register, click it, and modify its value. Press Esc to cancel the new value.

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

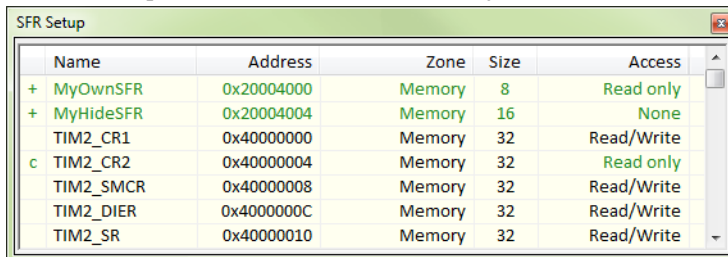
To change the display format, change the **Base** setting on the **Register Filter** page—available by choosing **Tools>Options**.

For the C-SPY Simulator and possibly in the C-SPY hardware debugger drivers, 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.

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

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 this window to customize the aspects of the SFRs. For factory-defined SFRs (that is, retrieved from the `ddf` file that is currently used), you can only customize the access type.

Any custom-defined SFRs are added to a dedicated register group called Custom, which you can choose to display in the **Register** window. Your custom-defined SFRs are saved in `projectCustomSFR.sfr`.

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

Requirements

None; this window is always available.

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 163.

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.

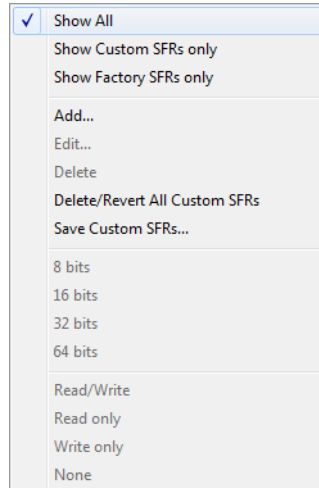
You can click a column header to sort the SFRs according to the column property.

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 186.

Edit

Displays the **Edit SFR** dialog box where you can edit an SFR, see *Edit SFR dialog box*, page 186.

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

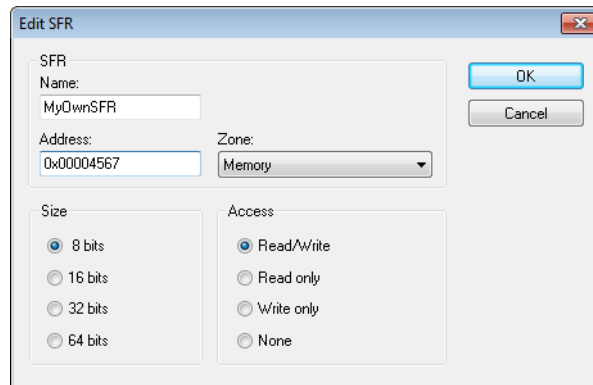
Selects display format for the selected SFR, which can be **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.

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 **SFR Setup** window.



Use this dialog box to define the SFRs.

Requirements

None; this dialog box is always available.

Name

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

Address

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

Zone

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

Size

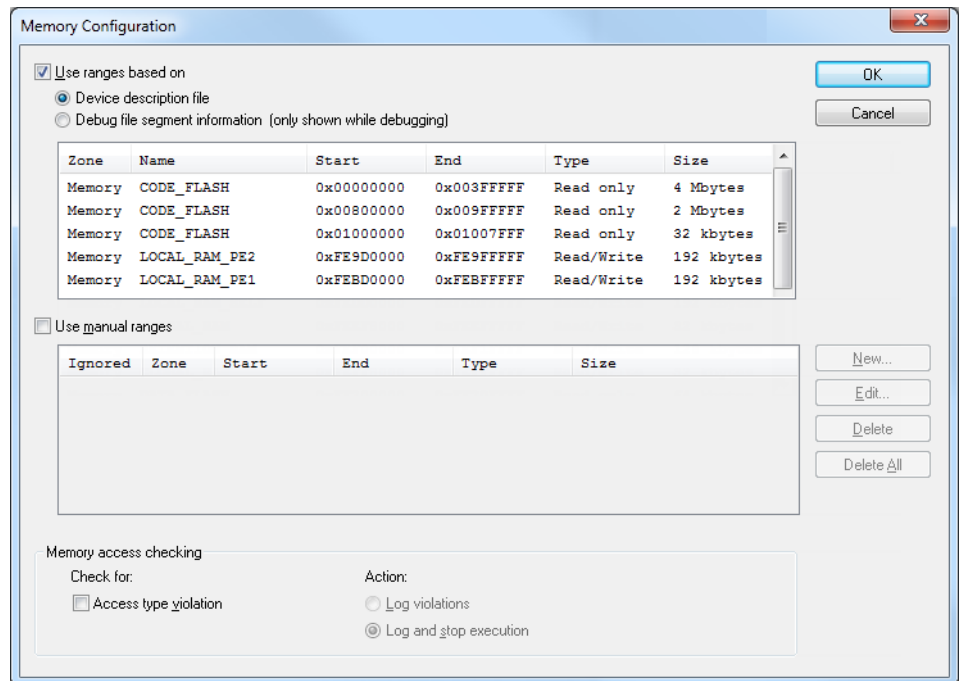
Selects the size of the SFR. Choose between **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.

Access

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

Memory Configuration dialog box, for the C-SPY simulator

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



To handle memory as efficiently as possible during debugging, C-SPY needs information about the memory configuration. By default, C-SPY uses a default

configuration based on information retrieved from the device description file that you have selected.

If that file does not specify memory ranges for the specific device that you are using, but instead for a family of devices (perhaps with varying amounts of on-chip RAM), use this dialog box to add memory areas so that they match the memory available on your device.

By default, the simulator does not allow accesses outside of the defined memory areas. In addition, you can use this dialog box to make C-SPY check that the access types are correct, which means that C-SPY will warn if there is a write access to read-only memory.

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 51.

This option is used by default.

Debug file segment information

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

Memory information is displayed in these columns:

Zone

The memory zone, see *C-SPY memory zones*, page 163.

Name

The name of the memory area.

Start

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

End

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

Type

The access type of the memory area.

Size

The size of the memory area.

Use manual ranges

Specify your own ranges manually via the **Edit Memory Range** dialog box. To open this dialog box, click **New** to specify a new memory range, or select a memory range and click **Edit** to modify it. For more information, see *Edit Memory Range dialog box, for the C-SPY simulator*, page 190.

The ranges you define manually are saved between debug sessions.

An **X** in the column **Ignored** means that the specified manual range is illegal, for example because it overlaps another range. Such an area will not be used.

Memory access checking

Check for determines what to check for:

- Access type violation.

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

- Log violations
- Log and stop execution.

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

Buttons

These buttons are available for the manual ranges:

New

Opens the **Edit Memory Range** dialog box, where you can specify a new memory range and associate an access type with it, see *Edit Memory Range dialog box, for the C-SPY simulator*, page 190.

Edit

Opens the **Edit Memory Range** dialog box, where you can edit the selected memory area. See *Edit Memory Range dialog box, for the C-SPY simulator*, page 190.

Delete

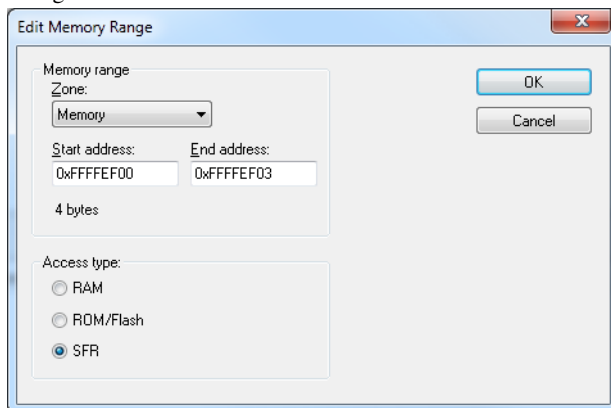
Deletes the selected memory area definition.

Delete All

Deletes all defined memory area definitions.

Edit Memory Range dialog box, for the C-SPY simulator

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



Use this dialog box to specify the memory areas, and their access types.

See also *Memory Configuration dialog box, for the C-SPY simulator*, page 187

Requirements

The C-SPY simulator.

Memory range

Defines the memory area specific to your device:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Start address

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

End address

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

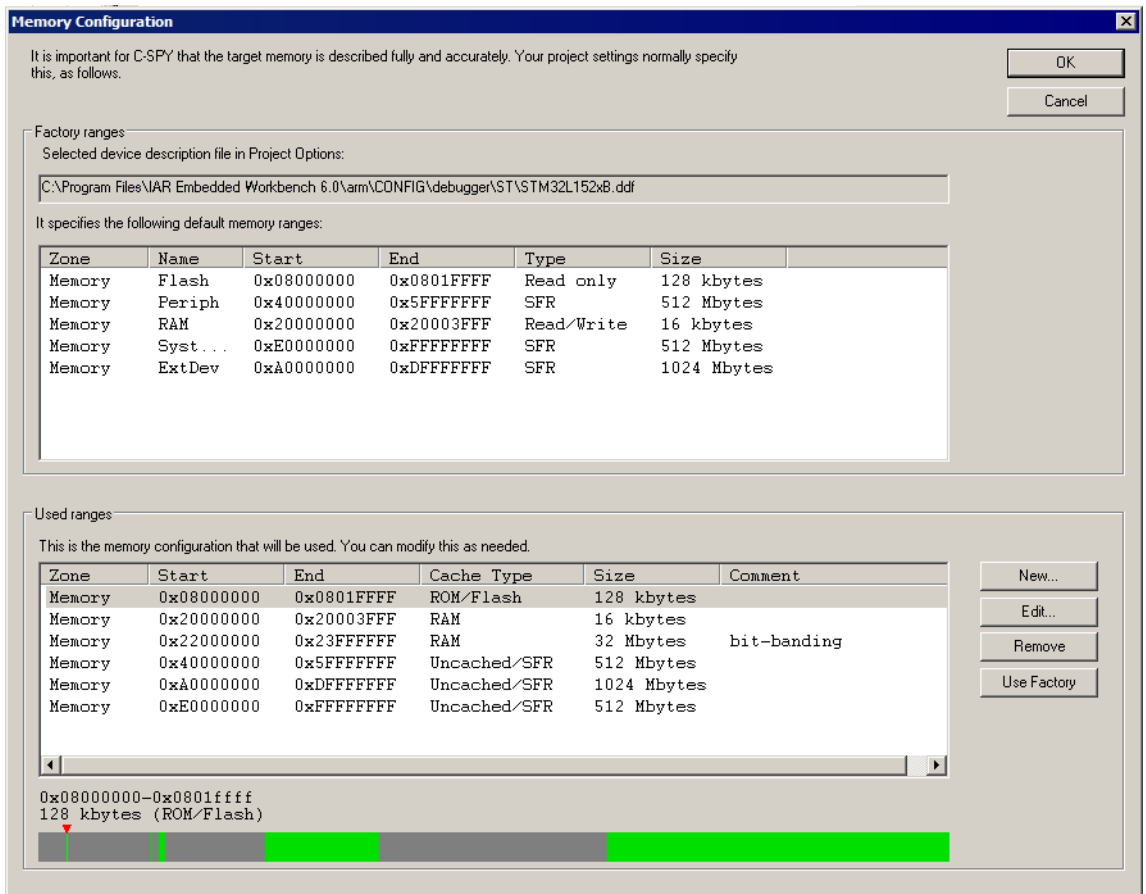
Access type

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

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

Memory Configuration dialog box, in C-SPY hardware debugger drivers

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



C-SPY uses a default memory configuration based on information retrieved from the device description file that you select, or if memory configuration is missing in the device description file, tries to provide a usable factory default. See *Selecting a device description file*, page 51.

Use this dialog box to verify, and if needed, modify the memory areas so that they match the memory available on your device. Providing C-SPY with information about the memory layout of the target system is helpful both in terms of performance and functionality:

- Reading (and writing) memory (if your debug probe is connected through a USB port) can be fast, but is usually the limiting factor when C-SPY needs to update many debugger windows. Caching memory can speed up the performance, but then C-SPY needs information about the target memory.
- If C-SPY has been informed that the content of certain memory areas will be changed during a debug session, C-SPY can keep a copy of that memory readable even when the target does not normally allow reading (such as when executing).
- C-SPY can prevent accesses to areas without any memory at all, which can be important for certain hardware.

The **Memory Configuration** dialog box is automatically displayed the first time you start the C-SPY driver for a given project, unless the device description file contains a memory description which is already specified as correct and complete. Subsequent starts will not display the dialog box unless you have made project changes that might cause the memory configuration to change, for example if you have selected another device description file.

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

Requirements

One of these alternatives:

- The C-SPY CMSIS-DAP driver
- The C-SPY I-jet/JTAGjet driver.

Factory ranges

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

Zone

The memory zone, see *C-SPY memory zones*, page 163.

Name

The name of the memory area.

Start

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

End

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

Type

The access type of the memory area.

Size

The size of the memory area.

Used ranges

These columns lists the memory areas that you have specified manually:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Start

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

End

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

Cache Type

The cache type of the memory area.

Size

The size of the memory area.

Comment

Memory area information.

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

Graphical bar

A graphical bar that visualizes the whole theoretical memory space for the device. Defined areas are highlighted in green.

Buttons

These buttons are available:

New

Opens the **Edit Memory Range** dialog box, where you can specify a new memory area and attach a cache type to it, see *Edit Memory Range dialog box, for C-SPY hardware debugger drivers*, page 194.

Edit

Opens the **Edit Memory Range** dialog box, where you can edit the selected memory area. See *Edit Memory Range dialog box, for C-SPY hardware debugger drivers*, page 194.

Remove

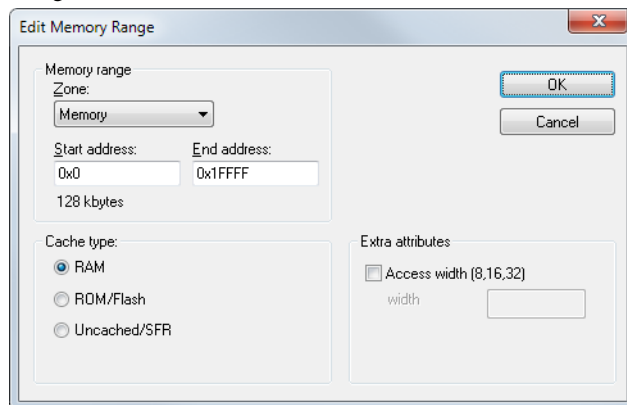
Removes the selected memory area definition.

Use Factory

Retrieves the memory areas as specified in the selected device description file, or if memory information is missing in the device description file, tries to provide a usable factory default.

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

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



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

See also *Memory Configuration dialog box, in C-SPY hardware debugger drivers*, page 191.

Requirements

One of these alternatives:

- The C-SPY CMSIS-DAP driver
- The C-SPY I-jet/JTAGjet driver.

Memory range

Defines the memory area specific to your device:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 163.

Start address

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

End address

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

Cache type

Selects a cache type to the memory area; choose between:

RAM

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

ROM/Flash

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

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

SFR/Uncached

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

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

Extra attributes

Provides extra attributes.

Access width [8,16,32]

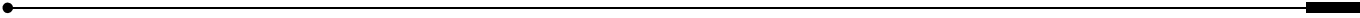
Forces C-SPY to use 8, 16, or 32 bits when accessing memory in this range. Specify 8, 16 or 32 in the text box.

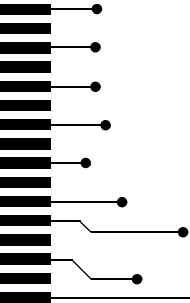
This option might not be available in the C-SPY driver you are using.

Part 2. Analyzing your application

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

- Trace
- Profiling
- 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 97
- *Getting started using event logging*, page 99
- *Power debugging*, page 275
- *Getting started using interrupt logging*, page 350
- *Profiling*, page 255

REASONS FOR USING TRACE

By using trace, you can inspect the program flow up to a specific state, for instance an application crash, and use the trace data to locate the origin of the problem. Trace data can be useful for locating programming errors that have irregular symptoms and occur sporadically.

Reasons for using the trace triggers and trace filters

By using trace trigger and trace filter conditions, you can select the interesting parts of your source code and use the trace buffer in the trace probe more efficiently. Trace triggers—Trace Start and Trace Stop breakpoints—specify for example a code section for which you want to collect trace data. A trace filter specifies conditions that, when fulfilled, activate the trace data collection during execution.

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

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

BRIEFLY ABOUT TRACE

Your target system must be able to generate trace data. Once generated, C-SPY can collect it and you can visualize and analyze the data in various windows and dialog boxes.

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

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

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

ETM trace

ETM (Embedded Trace Macrocell) real-time trace is a continuously collected sequence of every executed instruction for a selected portion of the execution. It is only possible to collect as much data as the trace buffer can hold. The trace buffer can be located either in the debug probe or on-chip (ETB). The trace buffer collects trace data in real time, but the data is not displayed in the C-SPY windows until after the execution has stopped.

PTM trace

PTM (Program Trace Macrocell) is an alternative implementation of the trace logic used in some ARM Cortex cores. The functionality is the same as ETM trace. Throughout this document, the term ETM also applies to PTM unless otherwise stated.

ETB trace

ETB (Embedded Trace Buffer) trace is an on-chip trace buffer. The trace buffer has a designated memory area with a predefined size.

MTB trace

MTB trace (Micro Trace Buffer) is a simplified variant of ETM trace, and uses an on-chip trace buffer. For MTB trace, the trace buffer shares the RAM memory with your application code.

MTB trace gives access to instruction trace on devices based on the Cortex-M0+ core.

SWO trace

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

- PC sampling

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

- Interrupt logs

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

- Data logs

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

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

If you use the SWO communication channel on a trace probe, the data will be collected in the trace buffer and displayed after the execution has stopped.

Trace features in C-SPY

In C-SPY, you can use the trace-related windows Trace, Function Trace, Timeline, and Find in Trace. In the C-SPY simulator, you can also use the Trace Expressions window. Depending on your C-SPY driver, you can set various types of trace breakpoints and triggers to control the collection of trace data.

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



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

currently using trace data. Clicking on the buttons opens the corresponding setup dialog boxes.

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

REQUIREMENTS FOR USING TRACE

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

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.

Requirements for using ETM trace

ETM trace is available for some ARM devices.

To use ETM trace you need one of these combinations:

- An I-jet, I-jet Trace, JTAGjet, or JTAGjet-Trace in-circuit debugging probe and a device that supports ETM via ETB. The debug probe reads ETM data from the ETB buffer. Make sure to use the C-SPY I-jet/JTAGjet driver.
- An I-jet Trace or JTAGjet-Trace in-circuit debugging probe and a device that supports ETM. Make sure to use the C-SPY I-jet/JTAGjet driver.
- A J-Trace debug probe and a device that supports ETM. Make sure to use the C-SPY J-Link/J-Trace driver.
- A J-Link or J-Trace debug probe and a device that supports ETM via ETB. The debug probe reads ETM data from the ETB buffer. Make sure to use the C-SPY J-Link/J-Trace driver.

For more information, see the *IAR Debug probes User Guide for I-jet, I-jet Trace, and I-scope*, the *JTAGjet-Trace User Guide for ARM* and the *IAR J-Link and IAR-J-Trace User Guide*, respectively.

Requirements for using MTB (Micro Trace Buffer) trace

To use MTB trace, you need one of these alternatives:

- An I-jet or JTAGjet in-circuit debugging probe and a device with MTB
- The C-SPY CMSIS-DAP driver and a device that has MTB and that supports CMSIS-DAP

Requirements for using SWO trace

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

Requirements for using the trace triggers and trace filters

The trace triggering and trace filtering features are available when ETM trace is available.

Collecting and using trace data

These tasks are covered:

- Getting started with ETM trace
- Getting started with SWO trace
- Setting up concurrent use of ETM and SWO
- Trace data collection using breakpoints
- Searching in trace data
- Browsing through trace data.

GETTING STARTED WITH ETM TRACE

1 Before you start C-SPY:

- For your device, the trace port must be set up. For some devices this is done automatically when the trace logic is enabled. However, for some devices, typically Atmel and ST devices based on ARM 7 or ARM 9, you need to set up the trace port explicitly. You do this by means of a C-SPY macro file. You can find examples of such files (`ETM_init*.mac`) in the example projects. To use a macro file, choose **Project>Options>Debugger>Setup>Use macro files**. Specify your macro file; a browse button is available for your convenience.

Note that the pins used on the hardware for the trace signals cannot be used by your application.

- 2 Start C-SPY and choose **ETM Trace Settings** from the C-SPY driver menu. In the **ETM Trace Settings** dialog box that appears, check if you need to change any of the default settings.



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

- 4 Click the **Edit Settings** button to open the **ETM Trace Settings** dialog box. Make sure that the ETM registers and pins were properly initialized and that the debug probe receives the Trace Clock (TCLK). The dialog box displays the trace clock frequency which is received by the debug probe. Click **Cancel** to close the dialog box.
- 5 Start the execution. When the execution stops, for instance because a breakpoint is triggered, trace data is displayed in the Trace window. For more information about the window, see *Trace window*, page 218.

GETTING STARTED WITH SWO TRACE

To get started using SWO trace:

- 1 Before you start C-SPY, choose **Project>Options>C-SPY driver**.
Click the **JTAG/SWD** tab or the **Connection** tab, respectively, and choose **Interface>SWD**. Alternatively, for I-jet, choose **JTAG** and the option **SWO>SWO on the TraceD0 pin**.
- 2 After you have started C-SPY, choose **SWO Trace Windows Settings** from the *C-SPY driver* menu. In the dialog box that appears, make your settings for controlling the output in the Trace window.

To see statistical trace data, select the option **Force>PC samples**, see *SWO Trace Window Settings dialog box*, page 212.
- 3 To configure the hardware's generation of trace data, click the **SWO Configuration** button available in the **SWO Configuration** dialog box. For more information, see *SWO Configuration dialog box*, page 214.

Note specifically these settings:

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



- 4 Open the SWO Trace window—available from the *C-SPY driver* menu—and click the **Activate** button to enable trace data collection.
- 5 Start the execution. The Trace window is continuously updated with trace data. For more information about this window, see *Trace window*, page 218.

SETTING UP CONCURRENT USE OF ETM AND SWO

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

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

TRACE DATA COLLECTION USING BREAKPOINTS

A convenient way to collect trace data between two execution points is to start and stop the data collection using dedicated breakpoints. Choose between these alternatives:

- In the editor or **Disassembly** window, position your insertion point, right-click, and toggle a **Trace Start** or **Trace Stop** breakpoint from the context menu.
- In the **Breakpoints** window, choose **Trace Start**, **Trace Stop**, or **Trace Filter**.
- The C-SPY system macros `__setTraceStartBreak` and `__setTraceStopBreak` can also be used.

For more information about these breakpoints, see *Trace Start breakpoints dialog box*, page 235 and *Trace Stop breakpoints dialog box*, page 236, respectively.

Using the trace triggers and trace filters:

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

SEARCHING IN TRACE DATA

When you have collected trace data, you can perform searches in the collected data to locate the parts of your code or data that you are interested in, for example, a specific interrupt or accesses of a specific variable.

You specify the search criteria in the **Find in Trace** dialog box and view the result in the **Find in Trace** window.

Note: The **Find in Trace** dialog box depends on the C-SPY driver you are using.

The **Find in Trace** window is very similar to the **Trace** window, showing the same columns and data, but *only* those rows that match the specified search criteria. Double-clicking an item in the **Find in Trace** window brings up the same item in the **Trace** window.

To search in your trace data:



- 1 On the **Trace** window toolbar, click the **Find** button.
- 2 In the **Find in Trace** dialog box, specify your search criteria.

Typically, you can choose to search for:

- A specific piece of text, for which you can apply further search criteria
- An address range
- A combination of these, like a specific piece of text within a specific address range.

For more information about the various options, see *Find in Trace dialog box*, page 251.

- 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 252.

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.

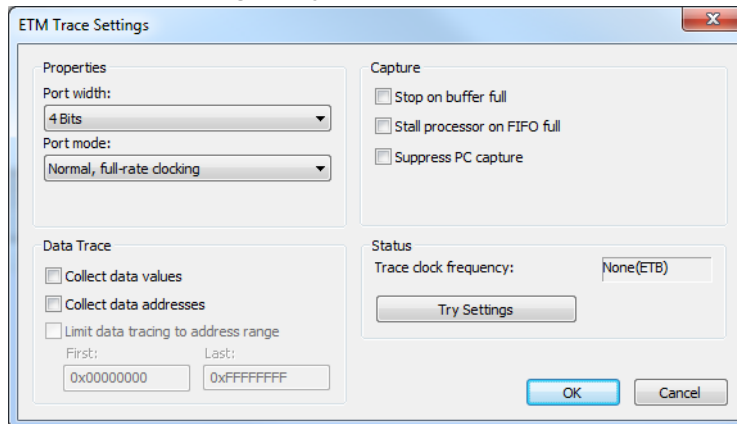
Reference information on trace

Reference information about:

- *ETM Trace Settings dialog box*, page 208
- *ETM Trace Settings dialog box (J-Link/J-Trace)*, page 210
- *SWO Trace Window Settings dialog box*, page 212
- *SWO Configuration dialog box*, page 214
- *Trace window*, page 218
- *Function Trace window*, page 223
- *Timeline window*, page 224
- *Viewing Range dialog box*, page 233
- *Trace Start breakpoints dialog box*, page 235 (simulator)
- *Trace Stop breakpoints dialog box*, page 236 (simulator)
- *Trace Start breakpoints dialog box (I-jet/JTAGjet and CMSIS-DAP)*, page 237
- *Trace Stop breakpoints dialog box (I-jet/JTAGjet and CMSIS-DAP)*, page 239
- *Trace Filter breakpoints dialog box (I-jet/JTAGjet)*, page 241
- *Trace Start breakpoints dialog box (J-Link/J-Trace)*, page 242
- *Trace Stop breakpoints dialog box (J-Link/J-Trace)*, page 245
- *Trace Filter breakpoints dialog box (J-Link/J-Trace)*, page 247
- *Trace Expressions window*, page 250
- *Find in Trace dialog box*, page 251
- *Find in Trace window*, page 252.
- *Trace Save dialog box*, page 253

ETM Trace Settings dialog box

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



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

See also:

- *Requirements for using ETM trace*, page 202
- *Getting started with ETM trace*, page 203.

Requirements

One of these alternatives:

- The C-SPY CMSIS-DAP driver
- The C-SPY I-jet/JTAGjet driver.

Port width

Specifies the trace bus width, which can be set to 1, 2, 4, 8, or 16 bits. The value must correspond with what is supported by the hardware and the debug probe.

For the lower values, the risk of FIFO buffer overflow increases, unless you are using the **Stall processor on FIFO full** option.

Port mode

Specifies the used trace clock rate:

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

- Demultiplexed
- Demultiplexed, half-rate clocking.

Data Trace

Selects what type of trace data you want C-SPY to collect. Choose between:

Collect data values

Collects data values.

Collect data addresses

Collects data addresses.

Limit data tracing to address range

Collects the specified type of data within the address range you specify in the **First** and **Last** text boxes.

Capture

Normally, trace collection starts or stops when execution starts or stops, or when a Trace Start or Trace Stop breakpoint is triggered. To change this, choose between:

Stop on buffer full

Stops collecting trace data when the probe buffer is full.

Stall processor on FIFO full

Stalls the processor in case the FIFO buffer fills up. The trace FIFO buffer on the CPU might in some situations become full—FIFO buffer overflow—which means trace data will be lost. This can be the case when the CPU is executing several branch instructions close to each other in time, such as in tight loops.

Suppress PC capture

Disables PC trace. Depending on your hardware, data trace might still be available.

Status

Shows the ETM status.

Trace clock frequency

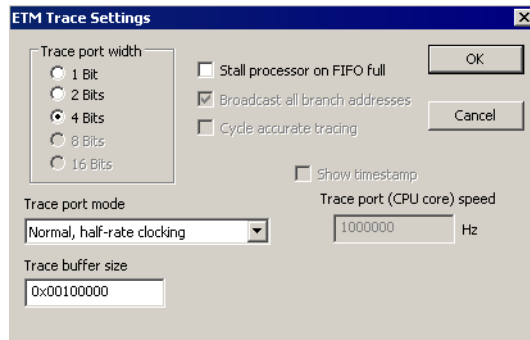
Shows the frequency of the trace clock to help you determine if the trace hardware is properly configured. Typically, this depends on the settings of Port Width and Port Mode.

Apply settings

Applies the settings you made in this dialog box. The trace clock frequency will be updated.

ETM Trace Settings dialog box (J-Link/J-Trace)

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



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

See also:

- *Requirements for using ETM trace*, page 202
- *Getting started with ETM trace*, page 203.

Requirements

The C-SPY J-Link/J-Trace driver.

Trace port width

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

For the lower values, the risk of FIFO buffer overflow increases, unless you are using the **Stall processor on FIFO full** option.

Trace port mode

Specifies the used trace clock rate:

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

Note: For the J-Trace driver, the available alternatives depend on the device you are using.

Trace buffer size

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

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

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

Cycle accurate tracing

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

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

Broadcast all branches

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

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

Stall processor on FIFO full

Stalls the processor in case the FIFO buffer fills up. The trace FIFO buffer on the CPU might in some situations become full—FIFO buffer overflow—which means trace data will be lost. This can be the case when the CPU is executing several branch instructions close to each other in time, such as in tight loops.

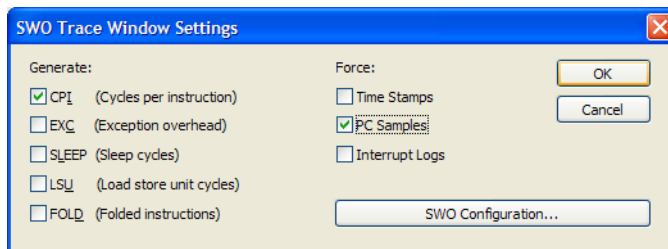
Show timestamp

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

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

SWO Trace Window Settings dialog box

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



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

Note that you also need to configure the generation of trace data, click **SWO Configuration**. For more information, see *SWO Configuration dialog box*, page 214.

Requirements

One of these alternatives:

- An I-jet or I-jet Trace in-circuit debugging probe
- A J-Link/J-Trace JTAG/SWD probe
- An ST-LINK JTAG/SWD probe.

Force

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

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

Time Stamps

Enables timestamps for various SWO trace packets, that is sent over the SWO communication channel. Use the resolution drop-down list to choose the resolution of the timestamp value. For example, 1 to count every cycle, or 16 to count every 16th cycle. Note that the lowest resolution is only useful if the time between each event packet is long enough. 16 is useful if using a low SWO clock frequency.

This option does not apply to I-jet.

PC samples

Enables sampling the program counter register, PC, at regular intervals. To choose the sampling rate, see *SWO Configuration dialog box*, page 214, specifically the option **PC Sampling**.

Interrupt Logs

Forces the generation of interrupt logs to the SWO Trace window. For information about other C-SPY features that also use trace data for interrupts, see *Interrupts*, page 343.

ITM Log

Forces the generation of ITM logs to the SWO Trace window.

This option applies to I-jet only.

Generate

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

CPI

Enables generation of trace data for the CPI counter.

EXC

Enables generation of trace data for the EXC counter.

SLEEP

Enables generation of trace data for the SLEEP counter.

LSU

Enables generation of trace data for the LSU counter.

FOLD

Enables generation of trace data for the FOLD counter.

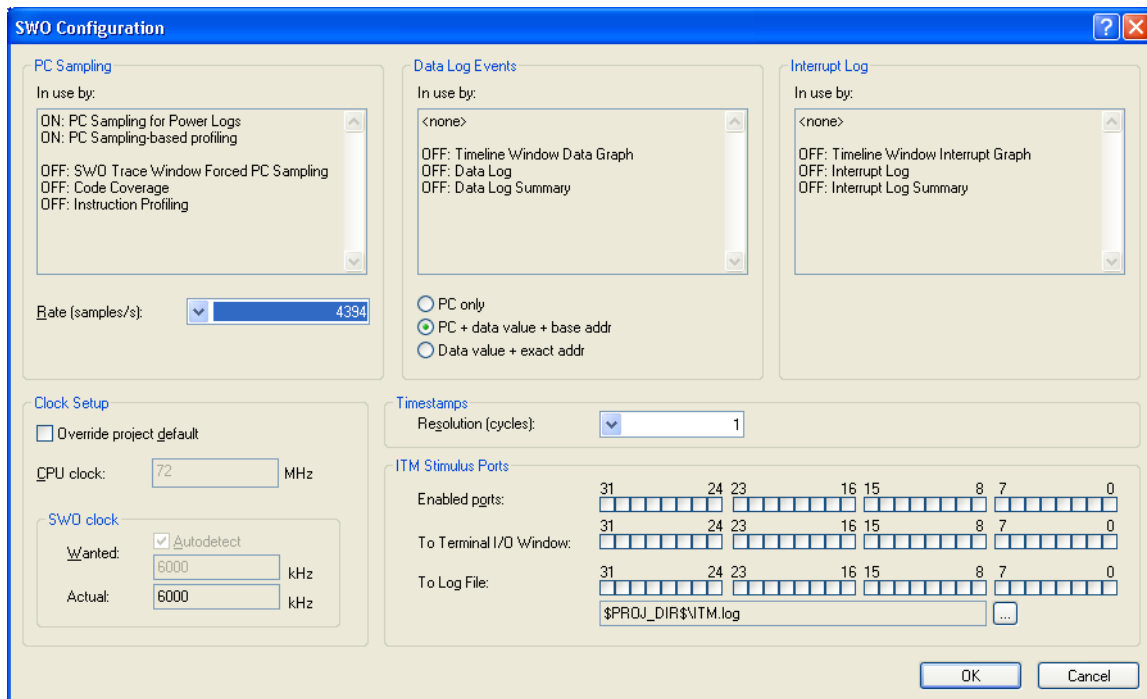
SWO Configuration

Displays the **SWO Configuration** dialog box where you can configure the hardware's generation of trace data. See *SWO Configuration dialog box*, page 214.

This button is not available when you are using I-jet.

SWO Configuration dialog box

The **SWO Configuration** dialog box is available from the C-SPY driver menu, alternatively from the **SWO Trace Window Settings** dialog box.



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

See also *Getting started with SWO trace*, page 204.

Requirements

One of these alternatives:

- An I-jet or I-jet Trace in-circuit debugging probe
- A J-Link/J-Trace JTAG/SWD probe
- An ST-LINK JTAG/SWD probe.

PC Sampling

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

In use by

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

Rate

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

This option does not apply to I-jet.

Divider

Select a divider, that, applied to the CPU clock speed, determines the rate of PC samples. The highest possible sampling rate depends on the SWO clock value and on how much other data that is sent over the SWO communication channel. The smaller values in the list will not work if the SWO communication channel is not fast enough to handle that much data.

This option applies to I-jet only.

Data Log Events

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

In use by

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

PC only

Logs the value of the program counter.

PC + data value + base addr

Logs the value of the program counter, the value of the data object, and its base address.

Data value + exact addr

Logs the value of the data object and the exact address of the data object that was accessed.

Interrupt Log

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

For more information about interrupt logging, see *Interrupts*, page 343.

Override project default

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

This option does not apply to I-jet.

Override project settings

Overrides the **CPU clock** and the **SWO prescaler** default values on the **Project>Options>I-jet>Setup** page.

This option only applies to I-jet.

CPU clock

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

This value is used for configuring the SWO communication speed.

For J-Link and ST-LINK, this value is also used for calculating timestamps.

SWO clock

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

Autodetect

Automatically uses the highest possible frequency that the J-Link debug probe can handle. When it is selected, the Wanted text box displays that frequency.

Wanted

Manually selects the frequency to be used, if **Autodetect** is not selected. The value can have decimals. Use this option if data packets are lost during transmission.

Actual

Displays the frequency that is actually used. This can differ a little from the wanted frequency.

This option does not apply to I-jet.

SWO prescaler

Specify the clock prescaler of the SWO communication channel. The prescaler, in turn, determines the SWO clock frequency. If data packets are lost during transmission, try using a higher prescaler value. Choose between:

Auto

Automatically uses the highest possible frequency that the I-jet debugging probe can handle.

1, 2, 5, 10, 20, 50, 100

The prescaler value.

This option applies to I-jet only.

Timestamps

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

This option does not apply to I-jet.

ITM Stimulus Ports

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

Enabled ports

Enables the ports to be used. Only enabled ports will actually send any data over the SWO communication channel to the debugger.

Port 0 is used by the terminal I/O library functions.

Ports 1-4 are used by the ITM macros for the Event Log window.

Port 5 is used for an optional PC value added to the ITM macro.

To Terminal I/O window

Specifies the ports to use for routing data to the Terminal I/O window.

To Log File

Specifies the ports to use for routing data to a log file. To use a different log file than the default one, use the browse button.



The `stdout` and `stderr` of your application can be routed via SWO to the C-SPY Terminal I/O window, instead of via semihosting. To achieve this, choose **Project>Options>General Options>Library Configuration>Library low-level**

interface implementation>stdout/stderr>Via SWO. This will significantly improve the performance of `stdout/stderr`, compared to when semihosting is used.

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

Trace window

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

This window displays the collected trace data.

The content of the **Trace** window depends on the C-SPY driver you are using and the trace support of your debug probe.

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

Requirements

One of these alternatives:

- The C-SPY Simulator
- An I-jet or I-jet Trace in-circuit debugging probe
- A JTAGjet debug probe
- A J-Link/J-Trace JTAG/SWD probe
- An ST-LINK JTAG/SWD probe.

Trace toolbar

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



Enable/Disable

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



Clear trace data

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



Toggle source

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



Browse

Toggles browse mode on or off for a selected item in the **Trace** window, see *Browsing through trace data*, page 206.

**Find**

Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 251.

**Save**

In the **ETM Trace** and **SWO Trace** windows this button displays the **Trace Save** dialog box, see *Trace Save dialog box*, page 253.

In the C-SPY I-jet/JTAG-jet driver and in the C-SPY simulator, this button displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.

**Edit Settings**

In the C-SPY simulator, this button is not enabled.

In the **ETM Trace** window this button displays the **Trace Settings** dialog box, see *ETM Trace Settings dialog box (J-Link/J-Trace)*, page 210 and *ETM Trace Settings dialog box*, page 208.

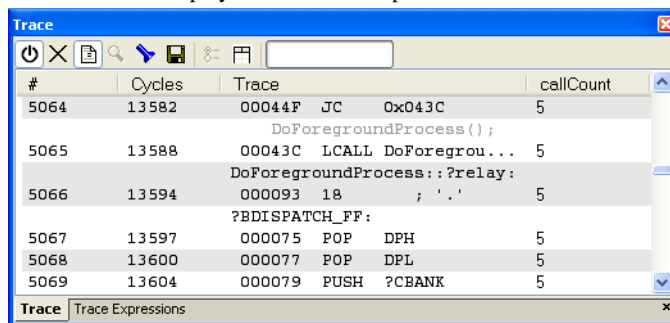
In the **SWO Trace** window this button displays the **SWO Trace Window Settings** dialog box, see *SWO Trace Window Settings dialog box*, page 212.

Edit Expressions (C-SPY simulator only)

Opens the **Trace Expressions** window, see *Trace Expressions window*, page 250.

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 for expressions.



#	Cycles	Trace	callCount
5064	13582	00044F JC 0x043C	5
		DoForegroundProcess();	
5065	13588	00043C LCALL DoForegrou...	5
		DoForegroundProcess::?relay:	
5066	13594	000093 1B ; '.'	5
		?BDISPATCH_FF:	
5067	13597	000075 POP DPH	5
5068	13600	000077 POP DPL	5
5069	13604	000079 PUSH ?CBANK	5

This area contains these columns for the C-SPY simulator:

#

A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.

Cycles

The number of cycles elapsed to this point.

Trace

The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.

Expression

Each expression you have defined to be displayed appears in a separate column. Each entry in the expression column displays the value *after* executing the instruction on the same row. You specify the expressions for which you want to collect trace data in the **Trace Expressions** window, see *Trace Expressions window*, page 250.

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 (for ETM trace)

This area contains these columns:

Index, #

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

Frame|Time

When collecting trace data in cycle-accurate mode (requires ARM7/9)—enable **Cycle accurate tracing** in the **ETM Trace Settings** dialog box—the value corresponds to the number of elapsed cycles since the start of the execution. This column is only available for the C-SPY J-Link/J-Trace driver.

When collecting trace data in non-cycle-accurate mode, the value corresponds to an approximate amount of cycles. For Cortex-M devices, the value is repeatedly calibrated with the actual number of cycles.

When the **Show timestamp** option is selected in the **ETM Trace Settings** dialog box, the value displays the time instead of cycles. To display the value as time requires collecting data in cycle-accurate mode, see *ETM Trace Settings dialog box*, page 208 and *ETM Trace Settings dialog box (J-Link/J-Trace)*, page 210 (specifically the **Cycle accurate tracing** option), and the J-Link/J-Trace driver.

Cycles

The number of cycles according to the internal JTAGjet-Trace timestamp.

This column is only available for JTAGjet-Trace.

Address

The address of the executed instruction.

Opcode

The operation code of the executed instruction.

This column is only available for J-Link/J-Trace.

Trace

The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.

Exec

The execution mode—ARM, Thumb, or NoExec.

This column is only available for JTAGjet-Trace.

Except

The type of exception, when it occurs.

This column is only available for JTAGjet-Trace.

Access

The data trace access type.

This column is only available for JTAGjet-Trace.

Data address

The data trace address.

This column is only available for JTAGjet-Trace.

Data value

The data trace value.

This column is only available for JTAGjet-Trace.

Comment

Additional information.

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 (for SWO trace)

This area contains these columns for SWO trace:

Index

An index number for each row in the trace buffer. Simplifies the navigation within the buffer.

This column is only available for JTAGjet-Trace.

SWO Packet

The contents of the captured SWO packet, displayed as a hexadecimal value.

Cycles

The approximate number of cycles from the start of the execution until the event.

For J-Link, this number is reported by the CPU.

For I-jet, this number corresponds to the internal I-jet/JTAGjet-Trace timestamp.

Event

The event type of the captured SWO packet. If the column displays `Overflow`, the data packet could not be sent, because too many SWO features use the SWO channel at the same time. To decrease the amount of transmissions on the communication channel, point at the SWO button—on the IDE main window toolbar—with the mouse pointer to get detailed tooltip information about which C-SPY features that have requested trace data generation. Disable some of the features.

Value

The event value, if any.

Trace

If the event is a sampled PC value, the disassembled instruction is displayed in this column. Optionally, the corresponding source code can also be displayed.

Comment

Additional information. This includes the values of the selected Trace Events counters, or the number of the comparator (hardware breakpoint) used for the Data Log breakpoint.

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.



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

Function Trace window

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

#	Cycles	Trace	myVariable
475	1050	0x000000E4: PutFib(unsigned int) + 76	1
476	1055	0x00000242: DoForegroundProcess() + 22	1
477	1058	0x0000025C: main() + 24	1
483	1069	0x0000022C: DoForegroundProcess()	1
485	1074	0x00000220: NextCounter()	1
491	1086	0x00000232: DoForegroundProcess() + 6	2
494	1092	0x0000007A: GetFib(int)	2
504	1109	0x0000023A: DoForegroundProcess() + 14	2

This window displays a subset of the trace data displayed in the **Trace** window. Instead of displaying all rows, the **Function Trace** window only shows trace data corresponding to calls to and returns from functions.

Requirements

One of these alternatives:

- The C-SPY Simulator
- An I-jet or I-jet Trace in-circuit debugging probe
- A JTAGjet debug probe
- A J-Link/J-Trace JTAG/SWD probe
- An ST-LINK JTAG/SWD probe.

Toolbar

For information about the toolbar, see *Trace window*, page 218.

Display area

For information about the columns in the display area, see *Trace window*, page 218

Timeline window

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

Depending on the abilities in hardware, the debug probe, and the C-SPY driver you are using, this window displays trace data in different graphs in relation to a common time axis:

- Call Stack graph
- Data Log graph
- Events graph
- Interrupt Log graph
- Power Log graph, see *Power graph in the Timeline window*, page 290.

To display a graph:

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

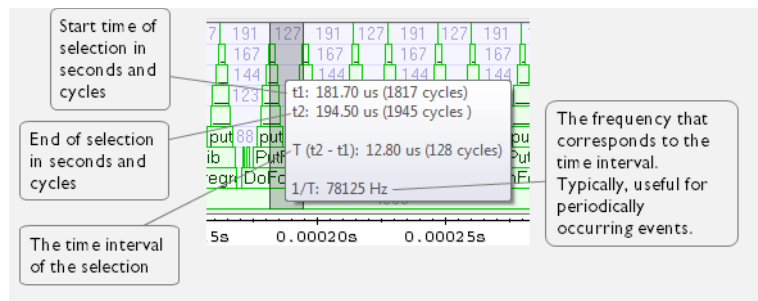
If you are using the C-SPY simulator you can ignore this step.

- 2** Choose **Timeline** from the C-SPY driver menu to open the **Timeline** window.
- 3** In the **Timeline** window, click in the graph area and choose **Enable** from the context menu to enable a specific graph.
- 4** For the Data Log graph, you need to set a Data Log breakpoint for each variable you want a graphical representation of in the **Timeline** window. See *Data Log breakpoints dialog box (C-SPY hardware drivers)*, page 151.
- 5** For the Event graph, you must add a preprocessor macro to your application source code where you want events to be generated. See *Getting started using event logging*, page 99.
- 6** Click **Go** on the toolbar to start executing your application. The graph appears.

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

- Right-click and from the context menu choose **Zoom In** or **Zoom Out**. Alternatively, use the + and - keys. The graph zooms in or out depending on which command you used.

- Right-click in the graph and from the context menu choose **Navigate** and the appropriate command to move backwards and forwards on the graph. Alternatively, use any of the shortcut keys: arrow keys, Home, End, and Ctrl+End.
- Double-click on a sample of interest and the corresponding source code is highlighted in the editor window and in the **Disassembly** window.
- Click on the graph and drag to select a time interval. Press Enter or right-click and from the context menu choose **Zoom>Zoom to Selection**. The selection zooms in. Point in the selection with the mouse pointer to get detailed tooltip information about the selected part of the graph:



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

Requirements

Depending on the abilities in hardware, the debug probe, and the C-SPY driver you are using, the display area can be populated with different graphs:

Target system	Call Stack graph	Data Log graph	Events graph	Interrupt Log graph	Power Log graph
C-SPY simulator	X	X	--	X	--
CMSIS-DAP	X ²	--	--	--	--
I-jet	X ²	X	X	X	X
JTAGjet	X ²	--	--	--	--
JTAGjet-Trace	X	--	--	--	--
J-Link	X ²	X	X	X	X
J-Trace	X	X ¹	X ¹	X ¹	--
ST-LINK	--	X	X	X	--

Table 11: Supported graphs in the Timeline window

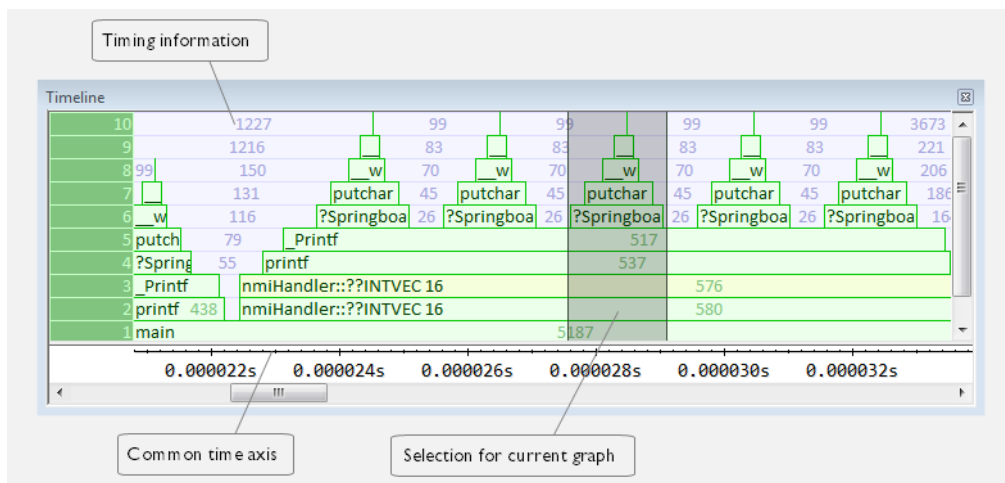
1 Very limited when ETM trace is enabled.

2 Requires ETB/MTB.

For more information about requirements related to trace data, see *Requirements for using trace*, page 202.

Display area for the Call Stack graph

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



At the bottom of the graph you will usually find `main`, and above it, the functions called from `main`, and so on. The horizontal bars, which represent invocations of functions, use four different colors:

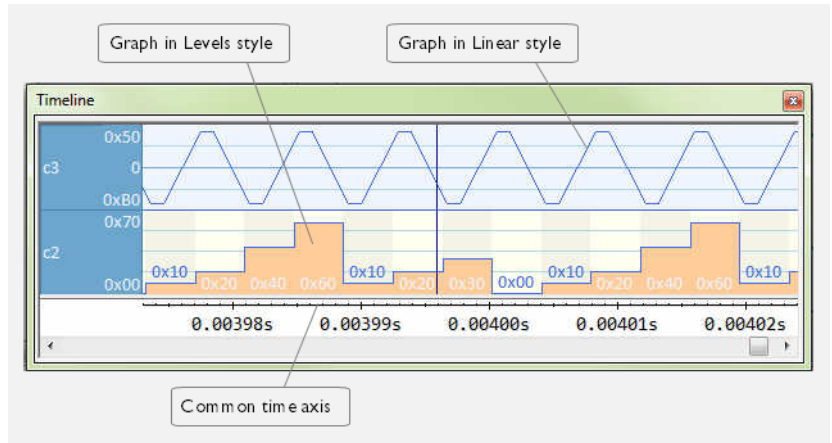
- Medium green for normal C functions with debug information
- Light green for functions known to the debugger only through an assembler label
- Medium or light yellow for interrupt handlers, with the same distinctions as for green.

The timing information represents the number of cycles spent in, or between, the function invocations.

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

Display area for the Data Log graph

The Data Log graph displays the data logs generated by SWO trace or by the C-SPY simulator, for up to four different variables or address ranges specified as Data Log breakpoints.



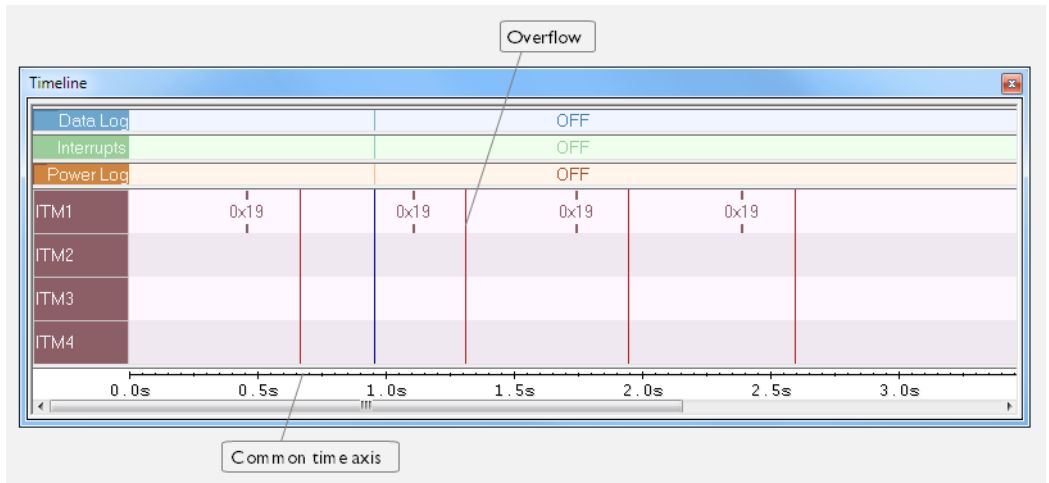
Where:

- The label area at the left end of the graph displays the variable name or the address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the Y-axis for a variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the **Data Log** window, see *Data Log window*, page 117.
- The graph can be displayed either as a thin line between consecutive logs or as a rectangle for every log (optionally color-filled).
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system. A red question mark indicates a log without a value.

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

Display area for the Events graph

The Events graph displays the events produced when the execution passes specific positions in your application code.



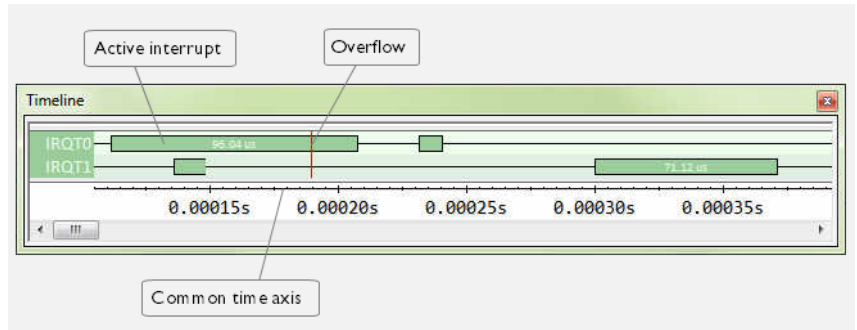
Where:

- The label area at the left end of the graph displays the name of the channel.
- For each channel, there will be a vertical line that indicates when the event occurred. Optionally, you can choose to display the event value that was passed with the event.
- The graph can be displayed as a thin line between consecutive logs, as a rectangle for every log (optionally color-filled), or as vertical bars.
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system.

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

Display area for the Interrupt Log graph

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



Where:

- The label area at the left end of the graph 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 357.
- 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.

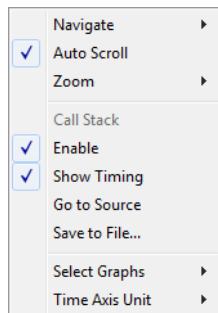
Selection and navigation

Click and drag to select. The selection extends vertically over all graphs, but appears highlighted in a darker color for the selected graph. You can navigate backward and forward in the selected graph using the left and right arrow keys. Use the Home and End

keys to move to the first or last relevant point, respectively. Use the navigation keys in combination with the Shift key to extend the selection.

Context menu

This context menu is available:



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

These commands are available:

Navigate (All graphs)

Commands for navigating over the graph(s); choose between:

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

Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll (All graphs)

Toggles auto scrolling on or off. When on, the most recently collected data is automatically displayed if you have executed the command **Navigate>End**.

Zoom (All graphs)

Commands for zooming the window, in other words, changing the time scale; choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +.

Zoom Out zooms out on the time scale. Shortcut key: -.

10ns, 100ns, 1us, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

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

10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Data Log (Data Log graph)

A heading that shows that the Data Log-specific commands below are available.

Events (Events graph)

A heading that shows that the Events-specific commands below are available.

Power Log (Power Log graph)

A heading that shows that the Power Log-specific commands below are available.

Call Stack (Call Stack graph)

A heading that shows that the Call stack-specific commands below are available.

Interrupt (Interrupt Log graph)

A heading that shows that the Interrupt Log-specific commands below are available.

Enable (All graphs)

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the **Timeline** window. If no trace data has been collected for a graph, **no data** will appear instead of the graph.

Show Timing (Call Stack graph)

Toggles the display of the timing information on or off.

Variable (Data Log graph)

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

Variable (Events graph)

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

Solid Graph (Data Log graph)

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

Viewing Range (Data, Event, and Power Log graph)

Displays a dialog box, see *Viewing Range dialog box*, page 233.

Size (Data, Event, and Power Log graph)

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

Style (Data, Event and Power Log graph)

Selects the style of the graph. Choose between:

Bars, displays a vertical bar for each log

Columns, displays a column for each log

Levels, displays the graph with a rectangle for each log, optionally color-filled

Linear, displays the graph as a thin line between consecutive logs

Note that all styles are not available for all graphs

Show Numerical Value (Data, Event, and Power Log graph)

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

Show Numbers (Events graph)

Shows the value of the event.

Hexadecimal (Events graph)

Determines the display mode for the value. Choose between hexadecimal or decimal. Note that this setting will also control the display mode for the same channel in the **Event Log** window and the **Event Log Summary** window.

Go To Source (Common)

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

Save to File (Call Stack graph)

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.

Open Setup Window (Power Log graph)

Opens the **Power Log Setup** window.

Sort by (Interrupt graph)

Sorts the entries according to their ID or name. The selected order is used in the graph when new interrupts appear.

Select Graphs (Common)

Selects which graphs to be displayed in the **Timeline** window.

Time Axis Unit (Common)

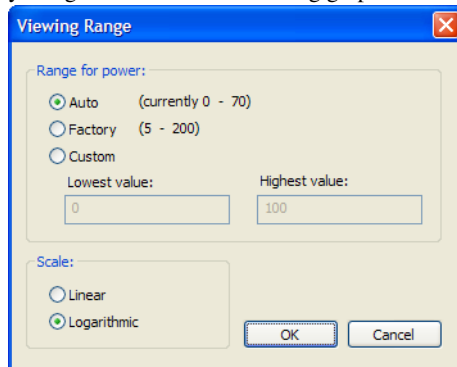
Selects the unit used in the time axis; choose between **Seconds** and **Cycles**.

Profile Selection

Enables profiling time intervals in the **Function Profiler** window. Note that this command is only available if the C-SPY driver supports PC Sampling.

Viewing Range dialog box

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



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

Requirements

One of these alternatives:

- The C-SPY Simulator
- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver.

Range for ...

Selects the viewing range for the displayed values:

Auto

Uses the range according to the range of the values that are actually collected, continuously keeping track of minimum or maximum values. The currently computed range, if any, is displayed in parentheses. The range is rounded to reasonably *even* limits.

Factory

For the Data Log graph: Uses the range according to the value range of the variable, for example 0–65535 for an unsigned 16-bit integer.

For the Power Log graph: Uses the range according to the properties of the measuring hardware.

Custom

Use the text boxes to specify an explicit range.

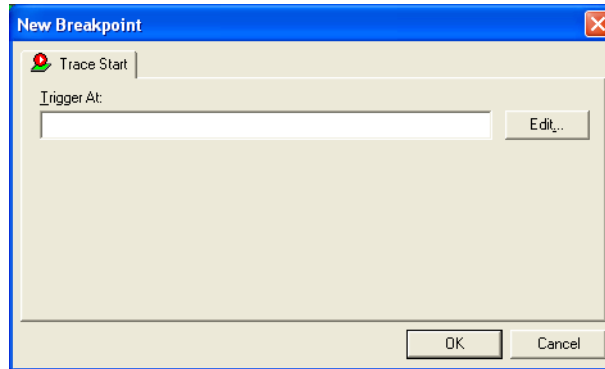
Scale

Selects the scale type of the Y-axis:

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

Trace Start breakpoints dialog box

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



Use this dialog box to set a Trace Start breakpoint where you want to start collecting trace data. If you want to collect trace data only for a specific range, you must also set a Trace Stop breakpoint where you want to stop collecting data.

See also, *Trace Stop breakpoints dialog box*, page 236.

To set a Trace Start breakpoint:

- 1 In the editor or **Disassembly** window, right-click and choose **Trace Start** from the context menu.
Alternatively, open the **Breakpoints** window by choosing **View>Breakpoints**.
- 2 In the **Breakpoints** window, right-click and choose **New Breakpoint>Trace Start**.
Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.
- 3 In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 4 When the breakpoint is triggered, the trace data collection starts.

Requirements

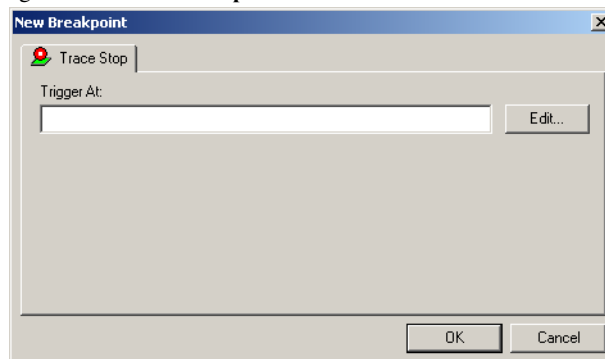
The C-SPY simulator.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 157.

Trace Stop breakpoints dialog box

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



Use this dialog box to set a Trace Stop breakpoint where you want to stop collecting trace data. If you want to collect trace data only for a specific range, you might also need to set a Trace Start breakpoint where you want to start collecting data.

See also, *Trace Start breakpoints dialog box*, page 235.

To set a Trace Stop breakpoint:

- 1 In the editor or **Disassembly** window, right-click and choose **Trace Stop** from the context menu.
Alternatively, open the **Breakpoints** window by choosing **View>Breakpoints**.
- 2 In the **Breakpoints** window, right-click and choose **New Breakpoint>Trace Stop**.
Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.
- 3 In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 4 When the breakpoint is triggered, the trace data collection stops.

Requirements

The C-SPY simulator.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 157.

Trace Start breakpoints dialog box (I-jet/JTAGjet and CMSIS-DAP)

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

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

Requirements

One of these alternatives:

- The C-SPY CMSIS-DAP driver
- The C-SPY I-jet/JTAGjet driver.

Trigger at

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

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.

Fetch

Accesses at execution address.

Any accesses of the specified type will activate the trace data collection.

Match data

Enables matching of the accessed data. Choose between:

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

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

Note: The **Match data** options are only available when using a Cortex-M device. For Cortex-M devices, only one breakpoint with **Match data** can be set. Such a breakpoint uses two breakpoint resources.

Size

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

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

Trigger range

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

Extend to cover requested range

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

This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.

Trace Stop breakpoints dialog box (I-jet/JTAGjet and CMSIS-DAP)

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

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

Requirements

One of these alternatives:

- The C-SPY CMSIS-DAP driver
- The C-SPY I-jet/JTAGjet driver.

Trigger at

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

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.

Fetch

Accesses at execution address.

Any accesses of the specified type will activate the trace data collection.

Match data

Enables matching of the accessed data. Choose between:

Value Specify a data value.

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

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

Note: The **Match data** options are only available when using a Cortex-M device. For Cortex-M devices, only one breakpoint with **Match data** can be set. Such a breakpoint uses two breakpoint resources.

Size

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

Auto Sets the size automatically. This can be useful if **Trigger at** contains a variable.

Manual Specify the size of the breakpoint range manually.

Trigger range

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

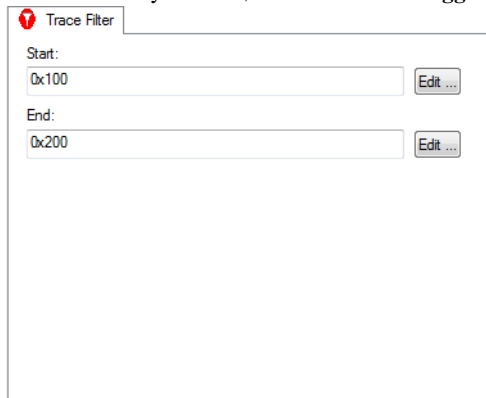
Extend to cover requested range

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

This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.

Trace Filter breakpoints dialog box (I-jet/JTAGjet)

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



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

Requirements

One of these alternatives:

- The C-SPY CMSIS-DAP driver
- The C-SPY I-jet/JTAGjet driver.

Start

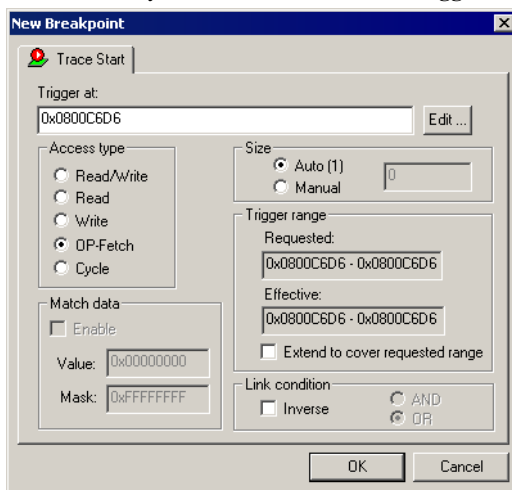
Specify the start location of the code section for which you want to collect trace data. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 157.

End

Specify the end location of the code section for which you want to collect trace data. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 157.

Trace Start breakpoints dialog box (J-Link/J-Trace)

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



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

Requirements

The C-SPY J-Link/J-Trace driver.

Trigger at

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

Size

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

Auto

Sets the size automatically. This can be useful if **Trigger at** contains a variable.

Manual

Specify the size of the breakpoint range manually.

Trigger range

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

Extend to cover requested range

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

This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.

Access Type

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.

OP-fetch

Accesses at execution address.

Cycle

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

Any accesses of the specified type will activate the trace data collection.

Match data

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

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

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

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

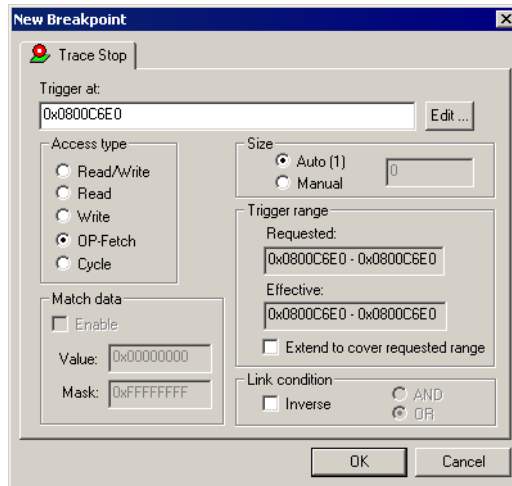
Link condition

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

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

Trace Stop breakpoints dialog box (J-Link/J-Trace)

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



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

Requirements

The C-SPY J-Link/J-Trace driver.

Trigger at

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

Size

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

Auto

Sets the size automatically. This can be useful if **Trigger at** contains a variable.

Manual

Specify the size of the breakpoint range manually.

Trigger range

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

Extend to cover requested range

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

This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.

Access Type

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.

OP-fetch

Accesses at execution address.

Cycle

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

Any accesses of the specified type will stop the trace data collection.

Match data

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

Value Specify a data value.

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

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

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

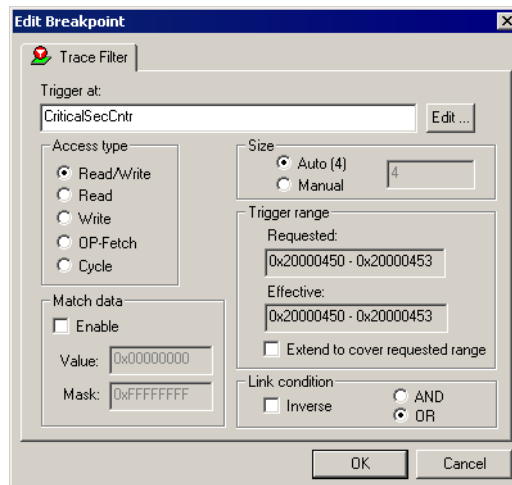
Link condition

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

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

Trace Filter breakpoints dialog box (J-Link/J-Trace)

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



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

Requirements

The C-SPY J-Link/J-Trace driver.

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 157.

Size

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

Auto

Sets the size automatically. This can be useful if **Trigger at** contains a variable.

Manual

Specify the size of the breakpoint range manually.

Trigger range

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

Extend to cover requested range

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

This option is not enabled for ARM7/9 devices because the range for such devices will always cover the whole data structure.

Access Type

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.

OP-fetch

Accesses at execution address.

Cycle

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

Match data

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

Value Specify a data value.

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

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

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

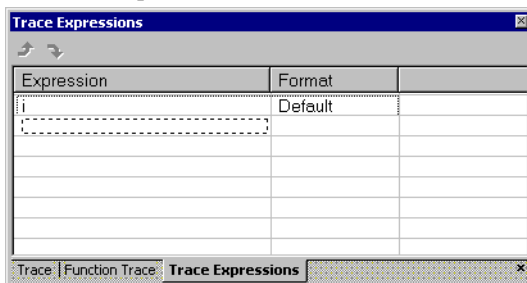
Link condition

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

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

Trace Expressions window

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



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

Requirements

The C-SPY simulator.

Toolbar

The toolbar buttons change the order between the expressions:

Arrow up

Moves the selected row up.

Arrow down

Moves the selected row down.

Display area

Use the display area to specify expressions for which you want to collect trace data:

Expression

Specify any expression that you want to collect data from. You can specify any expression that can be evaluated, such as variables and registers.

Format

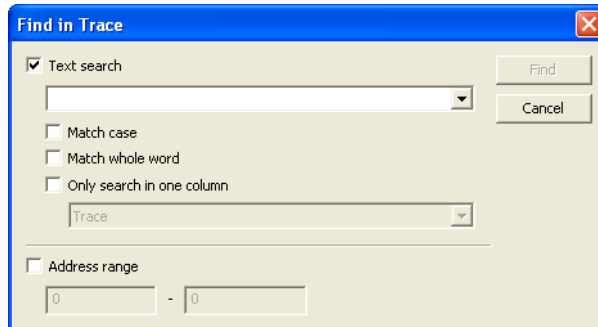
Shows which display format that is used for each expression. Note that you can change display format via the context menu.

Each row in this area will appear as an extra column in the Trace window.

Find in Trace dialog box

The **Find in Trace** dialog box is available by clicking the **Find** button on the **Trace** window toolbar or by choosing **Edit>Find and Replace>Find**.

Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the **Trace** window is the current window or the **Find** dialog box if the editor window is the current window.



Use this dialog box to specify the search criteria for advanced searches in the trace data.

The search results are displayed in the **Find in Trace** window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 252.

See also *Searching in trace data*, page 205.

Requirements

One of these alternatives:

- The C-SPY Simulator
- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY ST-LINK driver.

Text search

Specify the string you want to search for. To specify the search criteria, choose between:

Match Case

Searches only for occurrences that exactly match the case of the specified text. Otherwise **int** will also find **INT** and **Int** and so on.

Match whole word

Searches only for the string when it occurs as a separate word. Otherwise **int** will also find **print**, **sprintf** and so on.

Only search in one column

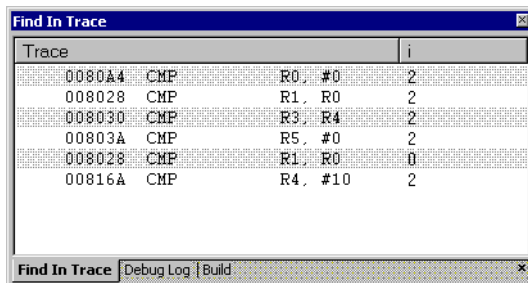
Searches only in the column you selected from the drop-down list.

Address Range

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

Find in Trace window

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



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 251.

For more information, see *Searching in trace data*, page 205.

Requirements

One of these alternatives:

- The C-SPY Simulator
- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver

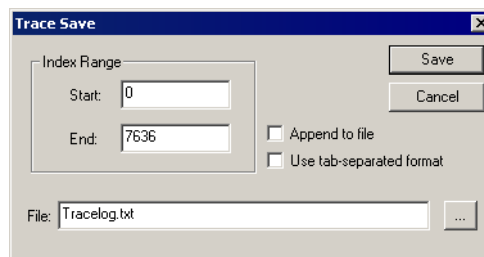
- The C-SPY ST-LINK driver.

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.

Trace Save dialog box

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



Requirements

One of these alternatives:

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

Index Range

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

Append to file

Appends the trace data to an existing file.

Use tab-separated format

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

File

Specify a file for the trace data.

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 ARM*.

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 (ETM trace) is analyzed to determine all function calls and returns. When the collected instruction sequence is incomplete or discontinuous, as sometimes happens when using ETM trace, the profiling information is less accurate. Select this profiling source (or Trace (flat)) to activate ETM trace for code coverage.
- Trace (flat) / Sampling

Each instruction in the full instruction trace (ETM trace) or each PC Sample (from SWO trace) is assigned to a corresponding function or code fragment, without regard to function calls or returns. This is most useful when the application does not exhibit normal call/return sequences, such as when you are using an RTOS, or when you are profiling code which does not have full debug information. Select this profiling source (or Trace (calls)) to activate ETM trace for code coverage.
- Breakpoints

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

Power sampling

Some debug probes support 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 support the profiler; there are no specific requirements.

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

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

This table lists the C-SPY driver profiling support:

Target system	Trace (calls)	Trace (flat)	Sampling	Power
C-SPY simulator	X	X	--	--
CMSIS-DAP	X	X	--	--
I-jet	X	X	X ¹	X
I-jet Trace	X	X	X ¹	X
JTAGjet/JTAGjet-Trace	X	X	--	--
J-Link	X	X	X ¹	--
J-Link Ultra	X	X	X ¹	X ²
J-Trace	X	X	X ¹	--
RDI	--	--	--	--
Macraigor	--	--	--	--
GDB Server	--	--	--	--
ST-LINK	--	--	X ¹	--
TI Stellaris	--	--	--	--
TI XDS	--	--	--	--
Angel	--	--	--	--
IAR ROM-monitor	--	--	--	--

Table 12: C-SPY driver profiling support

1 Only for Cortex-M devices supporting SWO.

2 Requires SWO trace.

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 13: Project options for enabling the profiler

- 2 To set up the profiler for function profiling:

- If you use ETM trace, make sure that the **Cycle accurate tracing** option is selected in the **Trace Settings** dialog box.
- If you use the SWD/SWO interface, no specific settings are required.



- 3 When you have built your application and started C-SPY, choose **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.

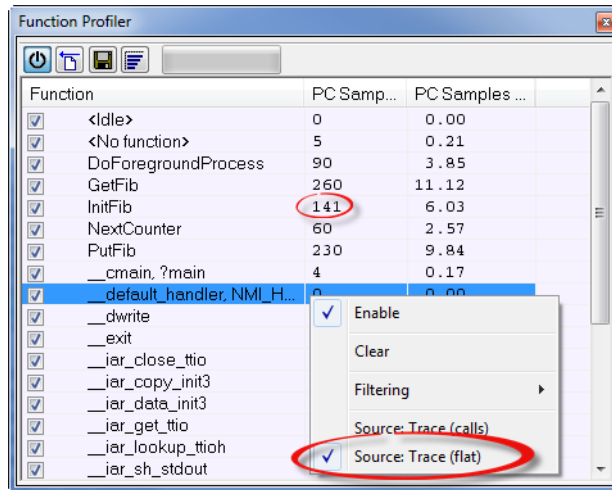
- For the **InitFib** function, **Flat Time** 231 is the time spent inside the function itself.
- For the **InitFib** function, **Acc Time** 487 is the time spent inside the function itself, including all functions **InitFib** calls.
- For the **InitFib/GetFib** function, **Acc Time** 256 is the time 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).

Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
main	1	165	3.58	4356	94.39
DoForegroundProcess	10			3704	
InitFib	1			487	
PutFib	10	3174	68.78	3174	68.78
NextCounter	10	100	2.17	100	2.17
InitFib	1	231	5.01	487	10.55
GetFib	16			256	
GetFib	26	416	9.01	416	9.01
DoForegroundProcess	10	270	5.85	3704	80.26
GetFib	10			160	
NextCounter	10				
PutFib	10				
<Other>	0	25			98.85
main	1				

The second figure shows the result of profiling using **Source: Trace (flat)**. In this case, the profiler does not follow the program flow, instead the profiler only detects whether the PC address is within the function scope. For incomplete trace data, the data might contain minor errors.

For the **InitFib** function, **Flat Time** 231 is the time (number of hits) spent inside the function itself.



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

GETTING STARTED USING THE PROFILER ON INSTRUCTION LEVEL

To display instruction profiling information in the Disassembly window:

- 1 When you have built your application and started C-SPY, choose **View>Disassembly** to open the **Disassembly** window, and choose **Instruction Profiling>Enable** from the context menu that is available when you right-click in the left-hand margin of the **Disassembly** window.
- 2 Make sure that the **Show** command on the context menu is selected, to display the profiling information.
- 3 Start executing your application to collect the profiling information.
- 4 When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, you can view instruction level profiling information in the left-hand margin of the window.

The screenshot shows a disassembly window titled "Disassembly" with a dropdown menu set to "main" and "Memory" selected. The assembly code is as follows:

```

Dly100us:
text_12:
0 08005F92 B082 SUB SP, SP, #0x8
Int32U Dly = (Int32U)arg;
0 08005F94 E005 B ??Dly100us_0
for(volatile int i = LOOP_DLY_100US; i; i--):
??Dly100us_1:
34 08005F96 9900 LDR R1, [SP]
5 08005F98 1E49 SUBS R1, R1, #0x1
5 08005F9A 9100 STR R1, [SP]
for(volatile int i = LOOP_DLY_100US; i; i--):
??Dly100us_2:
11 08005F9C 9900 LDR R1, [SP]
3 08005F9E 2900 CMP R1, #0x0
34 08005FA0 D1F9 BNE ??Dly100us_1
while(Dly--)
??Dly100us_0:
0 08005FA2 0001 MOVS R1, R0
0 08005FA4 1E48 SUBS R0, R1, #0x1

```

The left margin of the disassembly window contains execution counts for each instruction: 0, 0, 34, 5, 5, 11, 3, 34, 0, 0. These counts are highlighted by a vertical oval.

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

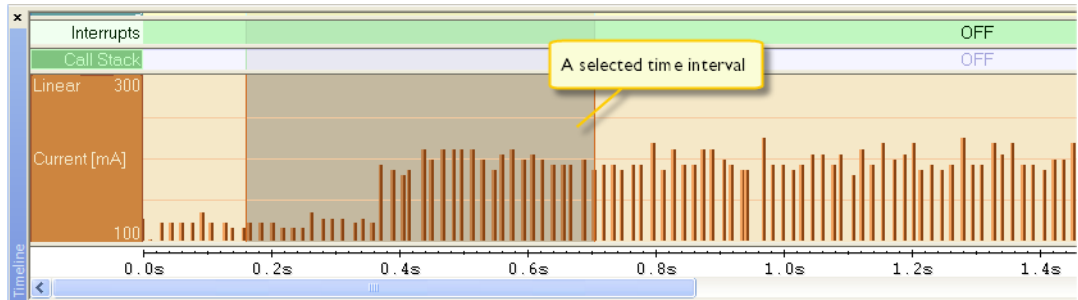
Instruction profiling attempts to use the same source as the function profiler. If the function profiler is not on, the instruction profiler will try to use first trace and then PC sampling as source. You can change the source to be used from the context menu that is available in the **Function Profiler** 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 I-jet and I-jet Trace in-circuit debugging probes, the JTAGjet debug probe, the J-Link probe, the J-Trace probe and the ST-LINK 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.

Function	PC Samples	PC Samples (%)	Power Samples	Energy (%)	Ax
GetButtons()	791	33.10	9	30.82	19
Dly100us(void *)	463	19.37	7	15.38	12
GLCD_SPI_TranserByte(Int3...	353	14.77	4	8.32	12
memcmp	325	13.60	4	14.64	21
main()	288	12.05	6	20.07	19
GLCD_Backlight(Int8U)	108	4.52	2	6.77	19
GLCD_SendCmd(GLCD_Cm...	43	1.80	0	0.00	-
GLCD_SPI_SendBlock(plnt8...	19	0.79	2	4.00	11
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 263

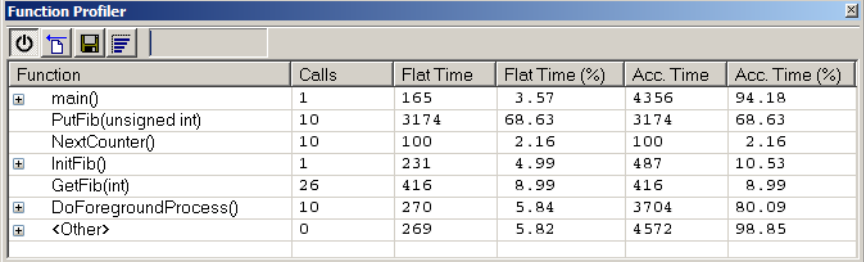
See also:

- *Disassembly window*, page 78
- *ETM Trace Settings dialog box (J-Link/J-Trace)*, page 210
- *ETM Trace Settings dialog box*, page 208
- *SWO Trace Window Settings dialog box*, page 212

- *SWO Configuration dialog box*, page 214

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"/> main()	1	165	3.57	4356	94.18
<input checked="" type="checkbox"/> PutFib(unsigned int)	10	3174	68.63	3174	68.63
<input checked="" type="checkbox"/> NextCounter()	10	100	2.16	100	2.16
<input checked="" type="checkbox"/> InitFib()	1	231	4.99	487	10.53
<input checked="" type="checkbox"/> GetFib(int)	26	416	8.99	416	8.99
<input checked="" type="checkbox"/> DoForegroundProcess()	10	270	5.84	3704	80.09
<input checked="" type="checkbox"/> <Other>	0	269	5.82	4572	98.85

This window displays function profiling information.

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

Requirements

One of these alternatives:

- The C-SPY Simulator
- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY ST-LINK driver.

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 256.

Status field

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

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 256.

Display area

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

- For the Breakpoints and Trace (calls) sources, the display area contains one line for each function compiled with debug information enabled. When some profiling information has been collected, it is possible to expand rows of functions that have called other functions. The child items for a given function list all the functions that have been called by the parent function and the corresponding statistics.
- For the Sampling and Trace (flat) sources, the display area contains one line for each C function of your application, but also lines for sections of code from the

runtime library or from other code without debug information, denoted only by the corresponding assembler labels. Each executed PC address from trace data is treated as a separate sample and is associated with the corresponding line in the Profiling window. Each line contains a count of those samples.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 256.

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

Function (All sources)

The name of the profiled C function.

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

Calls (Breakpoints and Trace (calls))

The number of times the function has been called.

Flat time (Breakpoints and Trace (calls))

The time expressed in cycles spent inside the function.

Flat time (%) (Breakpoints and Trace (calls))

Flat time expressed as a percentage of the total time.

Acc. time (Breakpoint and Trace (calls))

The time expressed in cycles spent inside the function and everything called by the function.

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

Accumulated time expressed as a percentage of the total time.

PC Samples (Trace (flat) and Sampling)

The number of PC samples associated with the function.

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

The number of PC samples associated with the function as a percentage of the total number of samples.

Power Samples (Power Sampling)

The number of power samples associated with that function.

Energy (%) (Power Sampling)

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

Avg Current [mA] (Power Sampling)

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

Min Current [mA] (Power Sampling)

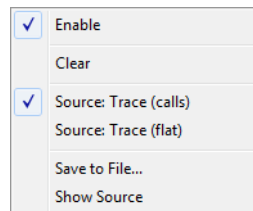
The minimum measured value for all samples associated with that function.

Max Current [mA] (Power Sampling)

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

Context menu

This context menu is available:



The contents of this menu depend on the C-SPY driver you are using.

These commands are available:

Enable

Enables the profiler. The system will collect information also when the window is closed.

Clear

Clears all profiling data.

Filtering

Selects which part of your code to profile. Choose between:

Check All—Excludes all lines from the profiling.

Uncheck All—Includes all lines in the profiling.

Load—Reads all excluded lines from a saved file.

Save—Saves all excluded lines to a file. Typically, this can be useful if you are a group of engineers and want to share sets of exclusions.

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

Source*

Selects which source to be used for the profiling information. See also *Profiling sources*, page 256. 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. This command is supported by the I-jet and I-jet Trace in-circuit debugging probes, the JTAGjet, the J-Link, and the J-Link Ultra debug probes.

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.

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

Code coverage

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

Introduction to code coverage

These topics are covered:

- Reasons for using code coverage
- Briefly about code coverage
- Requirements and restrictions for using code coverage.

REASONS FOR USING CODE COVERAGE

The code coverage functionality is useful when you design your test procedure to verify whether all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

BRIEFLY ABOUT CODE COVERAGE

The **Code Coverage** window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

REQUIREMENTS AND RESTRICTIONS FOR USING CODE COVERAGE

Code coverage is supported by the C-SPY Simulator and there are no specific requirements or restrictions.

To use code coverage in your hardware debugger system, consider these requirements and restrictions:

- When SWO trace is used: code coverage information is based on trace samples only. This means that a function must be executed several times before 100% code coverage is reached. Also, no code coverage information is collected while single stepping.

- When ETM trace is used: the only restriction is the size of the trace buffer. For efficient use of the trace buffer, you can limit the trace data collection using the trace start and trace stop breakpoints.

Reference information on code coverage

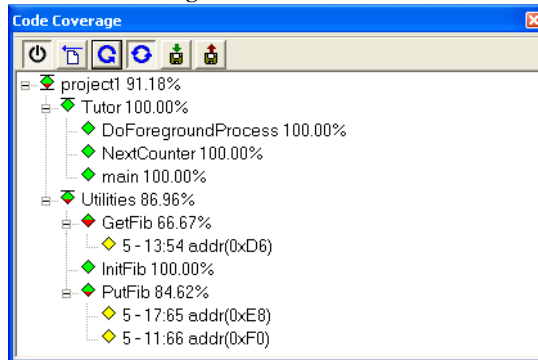
Reference information about:

- *Code Coverage window*, page 270.

See also *Single stepping*, page 72.

Code Coverage window

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



This window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

An asterisk (*) in the title bar indicates that C-SPY has continued to execute, and that the **Code Coverage** window must be refreshed because the displayed information is no longer up to date. To update the information, use the **Refresh** button.

To get started using code coverage:

- 1 Before using the code coverage functionality you must build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Include debug information in output
Debugger	Plugins>Code Coverage

Table 14: Project options for enabling code coverage

- 2 After you have built your application and started C-SPY, to activate ETM trace for code coverage, choose **C-SPY driver>Function Profiler** to open the **Function Profiler** window. Right-click in the window and choose **Trace (flat)** or **Trace (calls)** from the context menu. Then choose **View>Code Coverage** to open the **Code Coverage** window.



- 3 Click the **Activate** button, alternatively choose **Activate** from the context menu, to switch on code coverage.



- 4 Start the execution. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, click the **Refresh** button to view the code coverage information.

Requirements

One of these alternatives:

- The C-SPY Simulator
- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY ST-LINK driver.

Display area

The code coverage information is displayed in a tree structure, showing the program, module, function, and statement levels. The window displays only source code that was compiled with debug information. Thus, startup code, exit code, and library code is not displayed in the window. Furthermore, coverage information for statements in inlined functions is not displayed. Only the statement containing the inlined function call is marked as executed. The plus sign and minus sign icons allow you to expand and collapse the structure.

These icons give you an overview of the current status on all levels:

Red diamond	Signifies that 0% of the modules or functions has been executed.
Green diamond	Signifies that 100% of the modules or functions has been executed.
Red and green diamond	Signifies that some of the modules or functions have been executed.
Yellow diamond	Signifies a statement that has not been executed.

The percentage displayed at the end of every program, module, and function line shows the amount of statements that has been covered so far, that is, the number of executed statements divided with the total number of statements.

For statements that have not been executed (yellow diamond), the information displayed is the column number range and the row number of the statement in the source window, followed by the address of the step point:

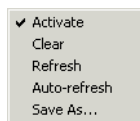
```
<column_start>-<column_end>: row address.
```

A statement is considered to be executed when one of its instructions has been executed. When a statement has been executed, it is removed from the window and the percentage is increased correspondingly.

Double-clicking a statement or a function in the **Code Coverage** window displays that statement or function as the current position in the editor window, which becomes the active window. Double-clicking a module on the program level expands or collapses the tree structure.

Context menu

This context menu is available:



These commands are available:



Activate

Switches code coverage on and off during execution.



Clear

Clears the code coverage information. All step points are marked as not executed.

**Refresh**

Updates the code coverage information and refreshes the window. All step points that have been executed since the last refresh are removed from the tree.

**Auto-refresh**

Toggles the automatic reload of code coverage information on and off. When turned on, the code coverage information is reloaded automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.

Save As

Saves the current code coverage result in a text file.

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 277.

BRIEFLY ABOUT POWER DEBUGGING

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

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

Measuring power consumption

The 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 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:

- A J-Link debug probe and a Cortex-M device with SWO. Note that the J-Link probe has very limited accuracy and a low resolution.
- An I-jet or I-jet Trace in-circuit debugging probe or a J-Link Ultra debug probe. Note that power debugging is not possible when using I-jet Trace with ETM.

Optimizing your source code for power consumption

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

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 core 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 core 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 core 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 also in the power domain, more optimizations can be made. The shadowed area represents the energy that could have been saved if the peripheral devices that are not used between t_2 and t_3 had been turned off, or if the priorities of the two tasks had been changed.

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

FINDING CONFLICTING HARDWARE SETUPS

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

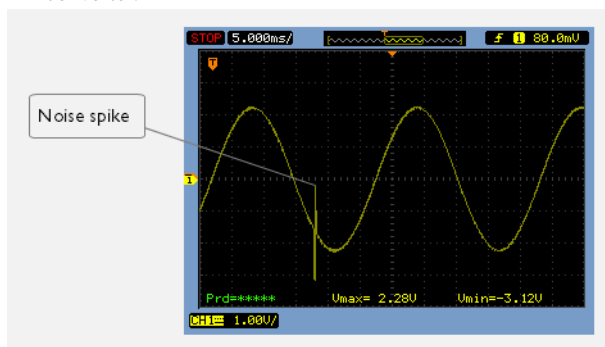
Timeline window. It is also possible to find the corresponding erratic initialization code by looking at the Power graph at application startup.

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

ANALOG INTERFERENCE

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

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



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*, page 224
- *Selecting a time interval for profiling information*, page 261.

DISPLAYING A POWER PROFILE AND ANALYZING THE RESULT

To view the power profile:

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

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

This step requires a Cortex-M3/M4 device.

- 2 Start the debugger.
- 3 Choose *C-SPY driver*>**Power Log Setup**. In the **ID** column, make sure to select the alternatives for which you want to enable power logging.
- 4 Choose *C-SPY driver*>**Timeline** to open the **Timeline** window.
- 5 Right-click in the graph area and choose **Enable** from the context menu to enable the power graph you want to view.
- 6 Choose *C-SPY driver*>**Power Log** to open the **Power Log** window.
- 7 Optionally, if you want to correlate power values to specific interrupts or variables, right-click in the Interrupts or Data Logs graph area, respectively, and choose **Enable** from the context menu.

For variables, you also need to set a Data Log breakpoint for each variable you want a graphical representation of in the **Timeline** window. See *Data Log breakpoints dialog box (C-SPY hardware drivers)*, page 151.

This step requires a Cortex-M3/M4 device.

- 8 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 233.
- 9 Click **Go** on the toolbar to start executing your application. In the **Power Log** window, all power values are displayed. In the **Timeline** window, you will see a graphical representation of the power values, and, if you are using a Cortex-M3/M4 device, of the data and interrupt logs if you enabled these graphs. For information about how to navigate on the graph, see *Timeline window*, page 224.

10 To analyze power consumption (requires a Cortex-M3/M4 device):

- 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 277.
- You can identify peripheral units to disable if they are not used. You can detect this by analyzing the power graph in combination with the other graphs in the **Timeline** window. See also *Detecting mistakenly unattended peripherals*, page 279.
- 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 261.

DETECTING UNEXPECTED POWER USAGE DURING APPLICATION EXECUTION

To detect unexpected power consumption:

- 1** Choose *C-SPY driver*>**SWO Configuration** to open the **SWO Configuration** dialog box. Make sure these settings are used:
 - **CPU clock** must be set to the same value as the CPU clock value set by your application. This is necessary to set the SWO clock and to obtain a correct data transfer to the debug probe.

This step requires a Cortex-M3/M4 device.

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

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

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

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:

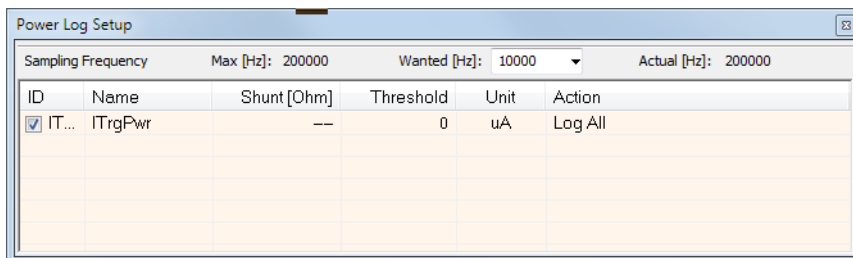
- *Power Log Setup window*, page 285
- *Power Log window*, page 287.
- *Power graph in the Timeline window*, page 290.

See also:

- *Trace window*, page 218
- *Timeline window*, page 224
- *Viewing Range dialog box*, page 233
- *Function Profiler window*, page 263.

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.

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

Requirements

One of these alternatives:

- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver.

Display area

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.

Shunt [Ohm]

This column always contains -- (two dashes) for all debug probes except I-scope.

For I-scope, specify the resistance of the shunt.

Threshold

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

Unit

Selects the unit for power display. Choose between: **nA**, **uA**, **mA**.

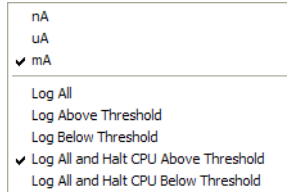
Action

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**

Context menu

This context menu is available:



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

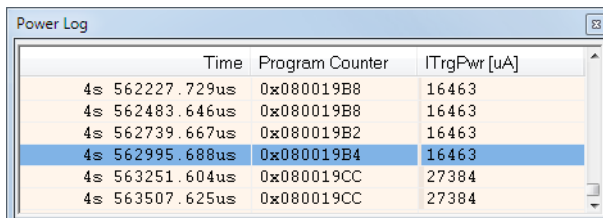
Logs all values. If a logged value exceeds the threshold, execution is stopped.

Log All and Halt CPU Below Threshold

Logs all values. If a logged value goes below the threshold, execution is stopped.

Power Log window

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



	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.

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

Note: The number of logged power values is limited. When this limit is exceeded, the entries at the beginning of the buffer are erased.

Requirements

One of these alternatives:

- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver.

Display area

This area contains these columns:

Time

The time from the application reset until the event, based on the clock frequency specified in the **SWO Configuration** dialog box.

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

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

Cycles

The number of cycles from the application reset until the event. 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

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.

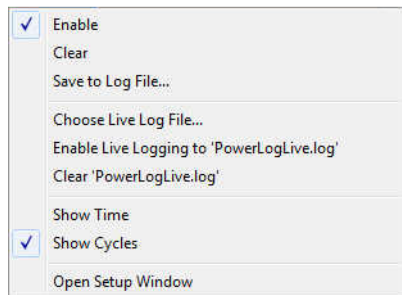
Idle, the power value is logged during idle mode.

Name [unit]

The power measurement value expressed in the unit you specified in the **Power Setup** window.

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.

Clear

Clears the power values saved internally within the IDE. The values will also be cleared when you reset the debugger, or if you change the execution frequency in the **SWO Configuration** dialog box.

Save to Log File

Displays a standard file selection dialog box where you can choose the destination file for the logged power values. This command then saves the current content of the internal log buffer.

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 in the **Power Log** window. This choice is also reflected in the log files.

Show Cycles

Displays the **Cycles** column in the **Power Log** window. This choice is also reflected in the log files.

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

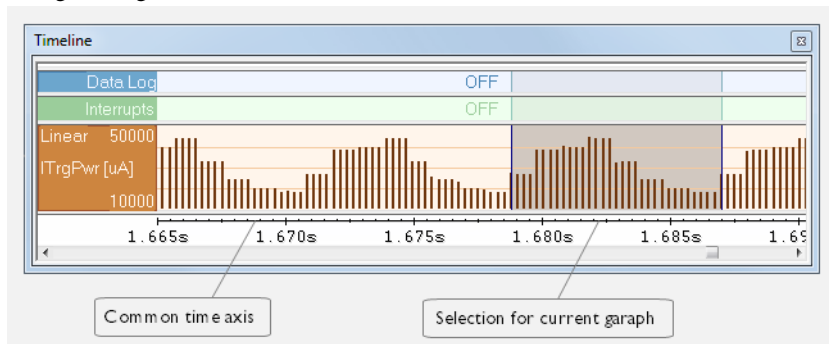
The value of the program counter close to the point where the power value was logged.

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.

Power graph in the Timeline window

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.

For more information about the **Timeline** window, how to display a graph, and the other supported graphs, see *Timeline window*, page 224.

See also *Requirements and restrictions for power debugging*, page 276.

Requirements

One of these alternatives:

- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver.

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.

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 ARM.

- 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 that C-RUN depends on the ARM semihosting 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 296.

For information about how to detect the errors, see *Detecting various runtime errors*, page 299.

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 320.
- The **C-RUN Message Rules** window, which lists all rules. See *C-RUN Messages Rules window*, page 322. 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.

If you instead want to use your own communication channel between your application and the host for C-RUN messages, replace the function `__iar_ReportCheckFailed` (uses the semihosting interface for the communication) with your own version and you can use any communication interface you like. In the source file `ReportCheckFailedStdout.c` (`arm\src\lib\crun`) you can find a variant that reports to the application's `stdout`. To use your own report function instead of the semihosting one, use the linker option `--redirect`
`__iar_ReportCheckFailed=__iar_ReportCheckFailedStdout`.

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 330.

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**.

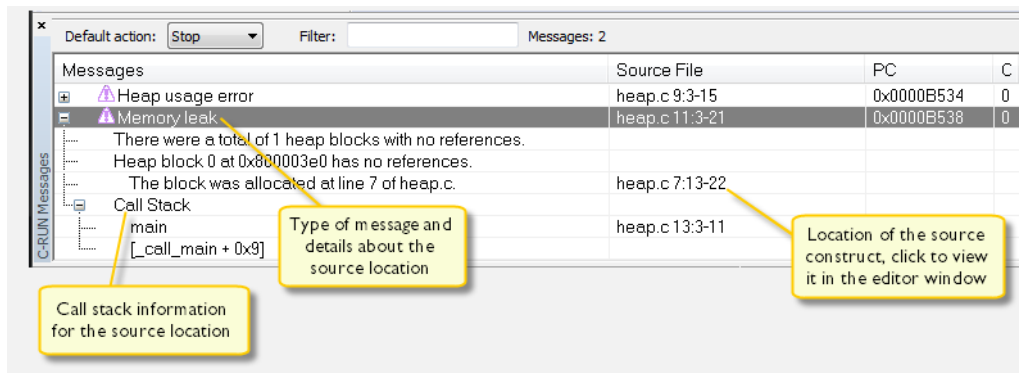
Note that runtime checking must be enabled on the project level, then you must enable each type of check you want to use. Some of the check options, such as **Use checked heap**, and **Enable bounds checking**, must be enabled on the project level, whereas others can be enabled on project or file level.

- 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 299.

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 division by zero
- Detecting bit loss or undefined behavior when shifting
- 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

	<p>integer conversions, which is useful when the loss of data caused by explicit conversion is considered intentional.</p>
<p>How to use it</p>	<p>Compiler option: <code>--runtime_checking integer_conversion implicit_integer_conversion</code></p> <p>In the IDE: Project>Options>Runtime Checking>Integer conversion</p> <p>The check can be applied to one or more modules.</p> <p>The check can be avoided by inserting an explicit mask:</p> <pre>short f(int x) { return x & 0xFFFF; /* Will not report change of value */ }</pre>
<p>How it works</p>	<p>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.</p> <p>Note that increment/decrement operators (<code>++/--</code>) and compound assignments (<code>+=, -=,</code> etc) are checked as if they were written longhand (<code>var = var op val</code>).</p> <p>For example, both <code>++i</code> and <code>i += 1</code> are checked as if they were written <code>i = i + 1</code>. 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 <code>int</code> 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:</p> <pre>signed char a = 127; void f(void) { ++a; /* Conversion check error (128 -> -128) */ a -= 1; /* Conversion check error (-129 -> 127) */ }</pre> <p>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.</p>
<p>Example</p>	<p>Follow the procedure described in <i>Getting started using C-RUN runtime error checking</i>, page 297, but use the Integer conversion option.</p>

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
Integer conversion failure	arith.c 12:8-14
Conversion changes the value from 500 (0x000001f4)	
to 244 (0x4).	
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 303.
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: <code>--runtime_checking signed_overflow unsigned_overflow</code> 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: <pre>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 */</pre>

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 297, 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
<ul style="list-style-type: none"> ▲ Signed integer overflow Result is greater than the largest representable number: 5 (0x5) + 2147483647 (0x7fffffff). Call Stack ovfl main [_call_main + 0x9] 	<ul style="list-style-type: none"> arith.c 26:22-35 arith.c 26:17-36 arith.c 35:3-8

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 297, but use the Integer shift overflow option.</p> <p>This is an example of source code that will be identified during runtime:</p>

```
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
 Shift overflow Result is greater than the largest representable number: signed value 5 (0x5) doubled 31 time(s). Call Stack	arith.c 32:3-10
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	<p>Compiler option: <code>--runtime_checking division_by_zero</code></p> <p>In the IDE: Project>Options>Runtime Checking>Division by zero</p> <p>The check can be applied to one or more modules.</p>
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	<p>Follow the procedure described in <i>Getting started using C-RUN runtime error checking</i>, page 297, but use the Division by zero option.</p> <p>This is an example of source code that will be identified during runtime:</p>

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 297, 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.	
Call 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* (linker option), page 324 (in the IDE, **Number of entries**). By default, 4-Kbyte entries that need about 190 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. 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 uninstrumented 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 `__as_make_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.

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 297, 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
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]	

Detecting heap usage error

Description

Checks that the heap interface—`malloc`, `new`, `free`, etc—is used properly by your application. The following improper uses are checked for:

- Using the incorrect deallocator—`free`, `delete`, etc—for an allocator—`malloc`, `new`, etc. For example:

```
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. */
```

- 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

Linker option: `--debug_heap`

In the IDE: **Project>Options>Runtime Checking>Use checked heap**

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.

The limit for how large a heap block can be at allocation is by default 1 Gbyte. The limit can be changed by the function:

```
size_t __iar_set_request_report_limit(size_t value);
```

The function returns the old limit. You can find the declaration of this function in `iar_dlmalloc.h`. For more information, see the *IAR C/C++ Development Guide for ARM*.

How it works

For any incorrect use of the heap interface, a message will be issued. See also *The checked heap provided by the library*, page 295.

Example

Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 297, 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
Heap usage error The address 0x800004a8 does not appear to be the start of a h...		heap.c 9:3-15
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

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.

How to use it

Linker option: `--debug_heap`

In the IDE: **Project>Options>Runtime Checking>Use checked heap**

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.

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 295.

Example

Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 297, 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
<ul style="list-style-type: none"> ▲ Memory leak 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. Call Stack main [_call_main + 0x9] 	<ul style="list-style-type: none"> heap_leak1.c:188:3-21 heap_leak1.c:185:13-22 heap_leak1.c:190:3-12

Detecting heap integrity violations

Description

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:

- 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:

```
char * p = (char *)malloc(100); /* Memory is allocated. */
...
p[100] = ... /* This write access is out of bounds. */
```

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.

- Write accesses to freed memory, for example:

```
char * p = (char *)malloc(...); /* Memory is allocated. */
...
free(p); /* Memory is freed. */
...
p[...] = ... /* Write access to freed memory. */
```

If the memory that contains the original `p` is allocated again before `p` is written to, this error will typically not be detected. By using the delayed free list (see below), this error can be found.

Why perform the check

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.

How to use it

Linker option: `--debug_heap`

In the IDE: **Project>Options>Runtime Checking>Use checked heap**

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.

For detecting heap integrity violations, you can use these functions which are defined in `iar_dlmalloc.h`:

- `size_t __iar_check_heap_integrity(void);`

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 295.

Example

Follow the procedure described in *Getting started using C-RUN runtime error checking*, page 297, 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		heap.c 10:3-30
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.		
Call Stack		heap.c 7:13-22
check		heap.c 11:1-1
main		heap.c 21:3-9
[_call_main + 0x9]		

Reference information on runtime error checking

Reference information about:

- *C-RUN Runtime Checking options*, page 318
- *C-RUN Messages window*, page 320
- *C-RUN Messages Rules window*, page 322

C-RUN Runtime Checking options

The **C-RUN Runtime Checking** options determine which checks to perform at runtime.

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 to track 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

When **Track pointer bounds** is enabled, pointers that originate from functions that are not instrumented for bounds checking are by default given fictive bounds information. Use this option to identify these pointers; any accesses via such pointers generate an error. In this way you can manually replace the fictive bounds information with valid counterparts; see *__as_get_base*, page 329, *__as_get_bound*, page 329, *__as_make_bounds*, page 330.

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.

Check pointers from non-instrumented memory

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 fictive bounds. Use this option to identify such pointers; any accesses via such pointers generate an error. In this way you can manually replace the fictive bounds information with valid counterparts; see *__as_get_base*, page 329, *__as_get_bound*, page 329, *__as_make_bounds*, page 330.

If this option is not used and you do not specify valid bounce information, accesses via such pointers do not generate errors and might result in unnoticed incorrect runtime behavior.

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

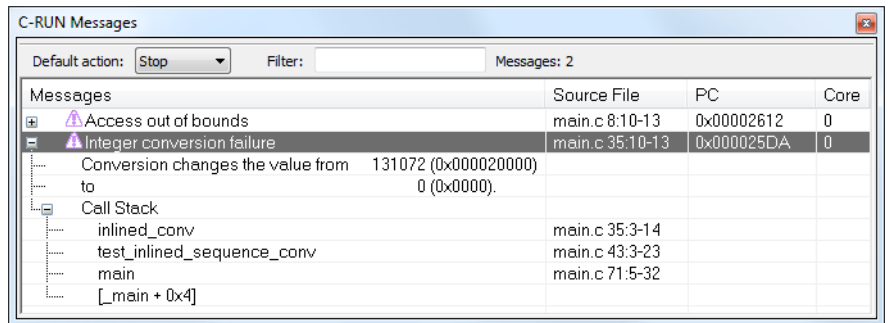
Checks for division by zero.

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.

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.

Display area

The display area shows all detected errors since the last reset.

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

Message

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.

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

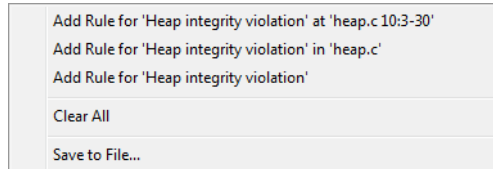
The value of PC when the runtime error was detected.

Core

The CPU core that executed the check, in case you have a multicore environment.

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.

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 wildcard.

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.

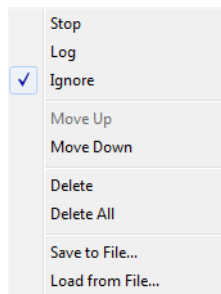
Action

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 332.

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` (linker option), page 324
- `--debug_heap` (linker option), page 325
- `--generate_entries_without_bounds`, page 325
- `--ignore_uninstrumented_pointers`, page 326
- `--ignore_uninstrumented_pointers` (linker option), page 326
- `--runtime_checking`, page 326
- `#pragma default_no_bounds`, page 327
- `#pragma define_with_bounds`, page 327
- `#pragma define_without_bounds`, page 328
- `#pragma disable_check`, page 328
- `#pragma generate_entry_without_bounds`, page 328
- `#pragma no_bounds`, page 329
- `__as_get_base`, page 329
- `__as_get_bound`, page 329
- `__as_make_bounds`, page 330
- `__as_make_bounds`, page 330

`--bounds_table_size` (linker option)


Syntax `--bounds_table_size records[:buckets] | (bytes)`

Parameters


`records` The number of records.

	<code>:buckets</code>	The number of buckets.
	<code>(bytes)</code>	The number of bytes, within parentheses.
Description	Use this linker option to specify the size of the global bounds table, which is used for tracking the bounds of pointers in memory.	
	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.	
	Alternatively, you can specify the total number of bytes to use for records and buckets.	
See also	<i>Detecting accesses outside the bounds of arrays and other objects</i> , page 305.	
		Project>Options>Runtime Checking>Number of entries

--debug_heap (linker option)

Syntax	<code>--debug_heap</code>	
Description	Use this linker option to use the checked heap.	
See also	<i>The checked heap provided by the library</i> , page 295.	
		Project>Options>Runtime Checking>Use checked heap

--generate_entries_without_bounds

Syntax	<code>--generate_entries_without_bounds</code>	
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 305.	
		Project>Options>Runtime Checking>Generate functions callable from non-instrumented code

--ignore_uninstrumented_pointers

Syntax `--ignore_uninstrumented_pointers`

Description Use this compiler option to disable checking of accesses via pointers from non-instrumented functions.

See also *Detecting accesses outside the bounds of arrays and other objects*, page 305.



Project>Options>Runtime Checking>Check pointers from non-instrumented functions

--ignore_uninstrumented_pointers (linker option)

Syntax `--ignore_uninstrumented_pointers`

Description Use this linker option to disable checking of accessing via pointers in memory for which no bounds have been set.

See also *Detecting accesses outside the bounds of arrays and other objects*, page 305.



Project>Options>Runtime Checking>Check pointers from non-instrumented memory

--runtime_checking

Syntax `--runtime_checking param ,param, ...`

Parameters

param is one of:

`signed_overflow |
unsigned_overflow`

Checks for signed or unsigned overflow in integer operations.

`integer_conversion |
implicit_integer_conversion`

Checks for implicit or explicit integer conversions resulting in a change of value.

`division_by_zero`


Checks for division by zero.

`signed_shift |
unsigned_shift`

Checks for bit loss or implementation-dependent results when shifting.

`switch`

Checks for unhandled cases in `switch` 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> .
Description	Use this compiler option to enable runtime error checking.	
See also	<i>Introduction to runtime error checking</i> , page 293.	
		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	<i>Detecting accesses outside the bounds of arrays and other objects</i> , page 305.	

#pragma define_with_bounds

Syntax	<code>#pragma define_with_bounds</code>	
Description	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.	
	This is useful for writing a checking version of a function based on the non-checking version.	

#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	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.

Description Use this operator to create a pointer of the same type as *ptr*, representing the upper bound of the area pointed to by *ptr*.

Example `bound = __as_get_bound(ptr);`

__as_make_bounds

Syntax `__as_make_bounds(ptr, number)`
`__as_make_bounds(ptr, base, bound)`

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 *ptr* up to *ptr + size* for *ptr*. 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 330
- `--rtc_output`, page 331
- `--rtc_raw_to_txt`, page 331
- `--rtc_rules`, page 332

--rtc_enable


Syntax `--rtc_enable`

	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 enable C-RUN run-time checking in <code>cspybat</code> . This option is automatically enabled if any of the other <code>-rtc_*</code> options are used.
	 This option is not available in the IDE.

--rtc_output

Syntax	<code>--rtc_output file</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>file</i> The file for output messages.
For use with	<code>cspybat</code>
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

`--rtc_rules file`

Note that this option must be placed before the `--backend` option on the command line.

Parameters

file The rules input file.

For use with

`cspybat`

Description

Use this option to specify the name of the C-RUN rules file to `cspybat`.

See also

C-RUN Messages Rules window, page 322 for information about **Save to File**.

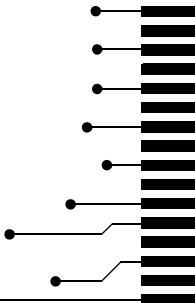


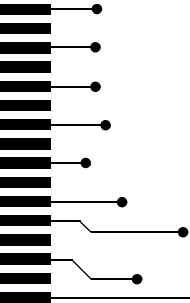
This option is not available in the IDE.

Part 3. Advanced debugging

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

- Multicore debugging
- Interrupts
- C-SPY macros
- The C-SPY command line utility—`cspybat`
- The flash loading mechanism





Multicore debugging

- Introduction to multicore debugging
- Debugging multiple cores
- Reference information on multicore debugging

Introduction to multicore debugging

These topics are covered:

- Briefly about application execution
- Symmetric multicore debugging
- Asymmetric multicore debugging
- Requirements and restrictions for multicore debugging

BRIEFLY ABOUT MULTICORE DEBUGGING

Multicore debugging means that you can debug targets with multiple cores. The C-SPY debugger supports multicore debugging in two ways:

- *Symmetric multicore debugging* (SMP), which means debugging two or more identical cores. This is handled using a single instance of the IAR Embedded Workbench IDE.
- *Asymmetric multicore debugging* (AMP), which means debugging two cores based on different architectures. It could be two different ARM-cores, for example a Cortex-A9 and a Cortex-M0. This is handled using two cooperating instances of the IAR Embedded Workbench IDE.

SYMMETRIC MULTICORE DEBUGGING

Symmetric multicore debugging means that the target has two or more identical cores on the board (usually on the same chip) that typically can be accessed through a single debug probe.

In the debugger, at any given time the windows show the state of only one of the cores—the one in focus.

This is an overview of special support for symmetric multicore debugging:

- You can control whether to automatically start and stop the whole application or to run the cores independently of each other.

- You can also control which core you want the debugger to focus on. This affects editor windows and the **Disassembly, Registers, Watch, Locals, Call Stack** window, etc.
- The **Cores** window shows a list of all available cores, and gives some information about each core, such as its execution state. The **Cores** toolbar is a complement to the **Cores** window,
- The **Stack** window can show the stack for each core by means of dedicated stack sections.
- RTOS support is available in separate multicore-aware plugins. Typically, they work like their single-core plugin counterparts, but handle multiple active tasks on separate cores. The plugins might also provide the information required by the **Stack** window to display the stack for any selected task.

ASYMMETRIC MULTICORE DEBUGGING

Asymmetric multicore debugging means that the target has two cores based on different architectures. Two IDE instances will be used, where each instance is connected to one core. The two IDE instances synchronize so that debugging sessions can be started and stopped and the cores can be controlled from either instance. Except for shared memory, each debugging session can only show information (variables, call stack, etc) about its own core.

You start one IDE instance manually and that instance is referred to as the *master*. When you start an asymmetric multicore debugging session, the master will initiate a second instance—the *slave*. The slave instance will be reused if it is already running.

The master and slave each require their own project. You have to set up each project with the correct processor variant, linker, and debugger options. The master project must also be configured to act as multicore master or have multicore master mode enabled.

One possible strategy for download is to combine the images for the cores into one and let the master project download the combined image. In this scenario, the slave must be configured to attach to a running target to suppress any downloading.

Another strategy is to download the master and slave as separate binary images, in which case you must make sure to avoid any unintentional overlaps in memory.

This is an overview of special support for asymmetric multicore debugging:

- You can control whether to automatically start and stop the whole application or to run the cores independently of each other.
- Each instance of the IDE displays debug information for the core that it is connected to.

- The **Cores** window shows a list of all available cores, and gives some information about each core, such as its execution state. The **Cores** toolbar is a complement to the **Cores** window,
- When you set a breakpoint it is connected to one core only and when the breakpoint is triggered, that core is stopped.

REQUIREMENTS AND RESTRICTIONS FOR MULTICORE DEBUGGING

The C-SPY simulator supports multicore debugging and there are no specific requirements or restrictions.

To use multicore debugging in your hardware debugger system, you need a specific combination of C-SPY driver and debug probe:

- The IAR C-SPY I-jet/JTAGjet driver
- An I-jet, I-jet Trace, JTAGjet, or JTAGjet-Trace debug probe

Note: There might be restrictions in trace support due to limitations in the hardware you are using.

Debugging multiple cores

These tasks are covered:

- Setting up for symmetric multicore debugging
- Setting up for asymmetric multicore debugging
- Starting and stopping a multicore debug session

SETTING UP FOR SYMMETRIC MULTICORE DEBUGGING

- 1 Choose **Project>Options>Debugger>Multicore** and specify the number of cores you have.
- 2 You can now start your debug session.

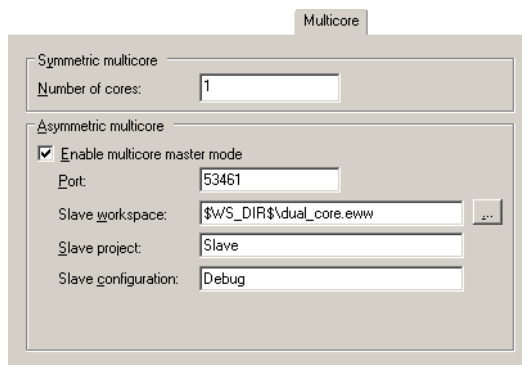
SETTING UP FOR ASYMMETRIC MULTICORE DEBUGGING

There are a number of ways that you can set up for multicore debugging, but this strategy is recommended:

- 1 Create a workspace with two projects, one for each core.

2 To configure the master project:

- Choose **Project>Options>Debugger>Multicore** and select **Enable multicore master mode**. Specify the workspace path, project name, and configuration name to use when starting the slave session.



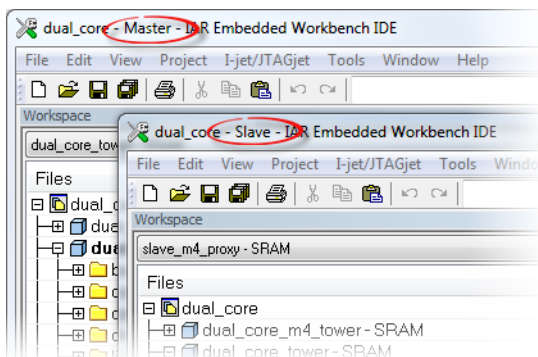
- Choose **Project>Options>C-SPY driver>Setup** and select a **Reset** strategy, typically **Hardware**.

3 To configure the slave project:

- Select that project in the workspace window, choose **Project>Options>Debugger>Download** and select **Attach to running target**.
- Choose **Project>Options>C-SPY driver>Setup** and select a **Reset** strategy that does not affect the master session, typically **Software**.

4 Make sure to use compatible settings for the debug probe for both projects.

5 The master and slave instances are indicated in the main window title bar.



STARTING AND STOPPING A MULTICORE DEBUG SESSION

- 1 To start a multicore debug session, for example use the standard **Download and Debug** command, either in the master or slave session.
- 2 To stop a multicore debug session, for example use the standard **Stop Debugging** command, which will stop both debugging sessions.

Reference information on multicore debugging

Reference information about:

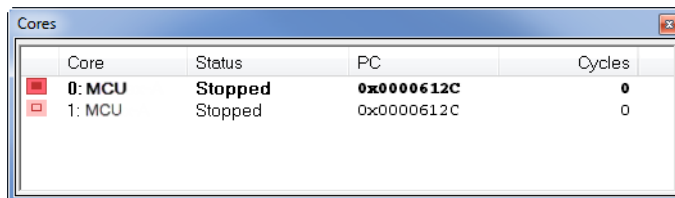
- *Cores window*, page 339
- *Cores toolbar*, page 341

See also:

- *__getSelectedCore*, page 392
- *__selectCore*, page 415

Cores window

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



Core	Status	PC	Cycles
0: MCU	Stopped	0x0000612C	0
1: MCU	Stopped	0x0000612C	0

This window shows a list of all available cores, and gives some information about each core, such as its execution state. The line highlighted in bold is the core currently in focus, which means that any window showing information that is specific to a core will be updated to reflect the state of the core in focus. This includes highlights in editor windows and the **Disassembly**, **Registers**, **Watch**, **Locals**, **Call Stack** window, etc. Double-click a line to focus on that core.

Note: For asymmetric multicore debugging, only the local core can be in focus.

If both cores are executing, and either one of them hits a breakpoint (or some other condition which causes the program execution to stop), then the debugger attempts to focus on that core automatically.

Requirements

One of these alternatives:

- The C-SPY simulator
- An I-jet, I-jet Trace, JTAGjet, or JTAGjet-Trace debug probe.

Display area

Each row in this area shows information about one of the cores, 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, in sleep mode



not in focus, in sleep mode

Core

The name of the core.

Status

The status of the execution, which can be one of **Stopped**, **Running**, or **Sleeping**.

PC

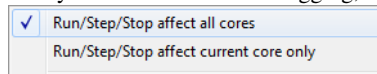
The value of the program counter.

Cycles | Time

The value of the cycle counter or the execution time since the start of the execution, depending on the debugger driver you are using.

Context menu

For symmetric multicore debugging, this context menu is available:



These commands are available:

Run/Step/Stop affect all cores

The **Run**, **Step**, **Stop** commands affect all cores.

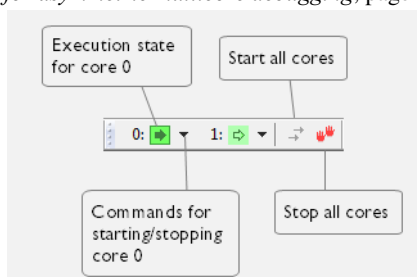
Run/Step/Stop affect current core only

The **Run/Step/Stop** commands affect only the current core. This menu command is only supported if your device supports it.

Note: These commands are not supported by all target hardware.

Cores toolbar

The **Cores** toolbar is available from the **View** menu if you have enabled multicore debugging, see *Setting up for symmetric multicore debugging*, page 337 or *Setting up for asymmetric multicore debugging*, page 337, respectively.



This toolbar is a complement to and shows the same state as the **Cores** window. Each core has a button with an adjacent drop-down menu. Click a button to make C-SPY focus on that core.

Note: For asymmetric multicore debugging, you can use the toolbar commands to start and stop cores in the associated debugging session.

Interrupts

- Introduction to interrupts
- Using the interrupt system
- Reference information on interrupts

Introduction to interrupts

These topics are covered:

- Briefly about interrupt logging
- Briefly about the interrupt simulation system
- Interrupt characteristics
- Interrupt simulation states
- C-SPY system macros for interrupt simulation
- Target-adapting the interrupt simulation system

See also:

- *Reference information on C-SPY system macros*, page 380
- *Breakpoints*, page 125
- *The IAR C/C++ Development Guide for ARM*

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. If you are using the C-SPY simulator, 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 program.

Requirements for interrupt logging

Interrupt logging is supported by the C-SPY simulator.

To use interrupt logging you need a Cortex-M device. You also need one of these alternatives:

- An I-jet or I-jet Trace in-circuit debugging probe or a JTAGjet debug probe, and an SWD interface between the debug probe and the target system
- A J-Link or J-Trace debug probe and an SWD interface between the debug probe and the target system
- An ST-LINK debug probe and an SWD interface between the debug probe and the target system

See also *Getting started using interrupt logging*, page 350.

BRIEFLY ABOUT THE INTERRUPT SIMULATION SYSTEM

By simulating interrupts, you can test the logic of your interrupt service routines and debug the interrupt handling in the target system long before any hardware is available. If you use simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices.

The C-SPY Simulator includes an interrupt simulation system where you can simulate the execution of interrupts during debugging. You can configure the interrupt simulation system so that it resembles your hardware interrupt system.

The interrupt system has the following features:

- Simulated interrupt support for the ARM core
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for various devices
- Configuration of hold time, probability, and timing variation
- State information for locating timing problems
- Configuration of interrupts using a dialog box or a C-SPY system macro—that is, one interactive and one automating interface. In addition, you can instantly force an interrupt.
- A log window that continuously displays events for each defined interrupt.
- A status window that shows the current interrupt activities.

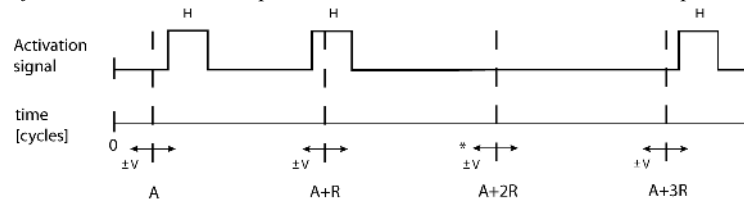
All interrupts you define using the **Interrupt Setup** dialog box are preserved between debug sessions, unless you remove them. A forced interrupt, on the other hand, exists only until it has been serviced and is not preserved between sessions.



The interrupt simulation system is activated by default, but if not required, you can turn off the interrupt simulation system to speed up the simulation. To turn it off, use either the **Interrupt Setup** dialog box or a system macro.

INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability*.



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

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

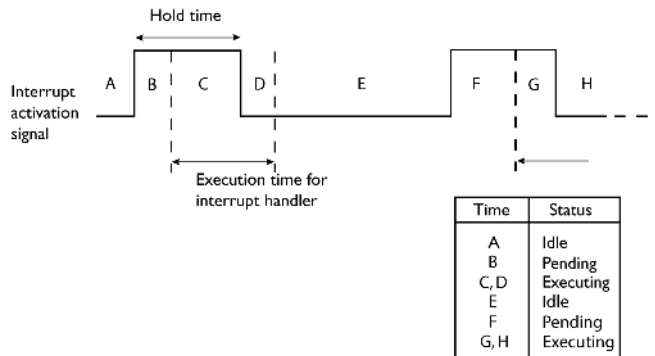
The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation 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

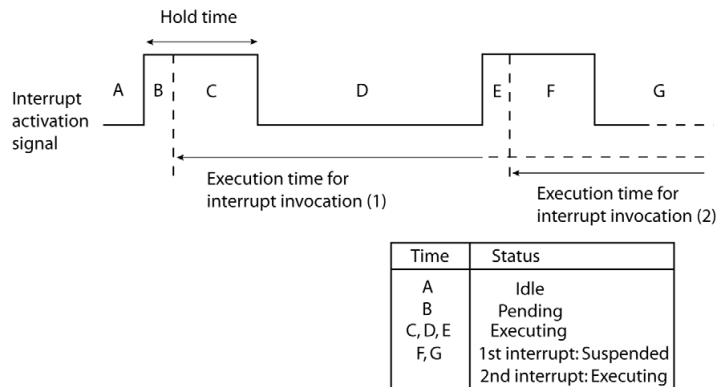
The interrupt simulation system contains status information that you can use for locating timing problems in your application. The **Interrupt Status** window displays the available status information. For an interrupt, these states can be displayed: *Idle*, *Pending*, *Executing*, or *Suspended*.

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:

```
__enableInterrupts
__disableInterrupts
__orderInterrupt
__cancelInterrupt
__cancelAllInterrupts
__popSimulatorInterruptExecutingStack
```

The parameters of the first five macros correspond to the equivalent entries of the **Interrupts** dialog box.

For more information about each macro, see *Reference information on C-SPY system macros*, page 380.

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 51.

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 367 for details about how to use a setup file to define simulated interrupts at C-SPY startup
- The tutorial *Simulating an interrupt* in the Information Center.

SIMULATING A SIMPLE INTERRUPT

This example demonstrates the method for simulating a timer interrupt for OKI ML674001. However, the procedure can also be used for other types of interrupts.

To simulate and debug an interrupt:

- I Assume this simple application which contains an IRQ handler routine that handles system timer interrupts. It increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```

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

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

unsigned int ticks = 0;

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

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

```

- 2 Add your interrupt service routine to your application source code and add the file to your project.
- 3 Build your project and start the simulator.
- 4 Choose **Simulator>Interrupt Setup** to open the **Interrupts Setup** dialog box. Select the **Enable interrupt simulation** option to enable interrupt simulation. Click **New** to open the **Edit Interrupt** dialog box. For the Timer example, verify these settings:

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

Table 15: Timer interrupt settings

Click **OK**.

- 5 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:
 - Generate an interrupt when the cycle counter has passed 4000
 - Continuously repeat the interrupt after approximately 2000 cycles.
- 6 To watch the interrupt in action, choose **Simulator>Interrupt Log** to open the Interrupt Log window.
- 7 From the context menu, available in the **Interrupt Log** window, choose **Enable** to enable the logging. If you restart program execution, status information about entrances and exits to and from interrupts will now appear in the **Interrupt Log** window.

For information about how to get a graphical representation of the interrupts correlated with a time axis, see *Timeline window*, page 224.

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 To set up for interrupt logging, choose *C-SPY driver*>**SWO Configuration**. In the dialog box, set up the serial-wire output communication channel for trace data. Note specifically the **CPU clock** option. The CPU clock can also be set up on the **Project>Options>ST-LINK** page.
For the C-SPY simulator, no specific settings are required.
- 2 Choose *C-SPY driver*>**Interrupt Log** to open the **Interrupt Log** window. Optionally, you can also choose:
 - *C-SPY driver*>**Interrupt Log Summary** to open the **Interrupt Log Summary** window
 - *C-SPY driver*>**Timeline** to open the **Timeline** window and view the Interrupt graph.
- 3 From the context menu in the **Interrupt Log** window, choose **Enable** to enable the logging.
In the **SWO Configuration** dialog box, you can see in the **Interrupt Log Events** area that interrupt logs are enabled.
- 4 Start executing your application program to collect the log information.
- 5 To view the interrupt log information, look in any of the Interrupt Log, Interrupt Log Summary, or the Interrupt graph in the **Timeline** window.
- 6 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
- 7 To disable interrupt logging, from the context menu in the **Interrupt Log** window, toggle **Enable** off.

Reference information on interrupts

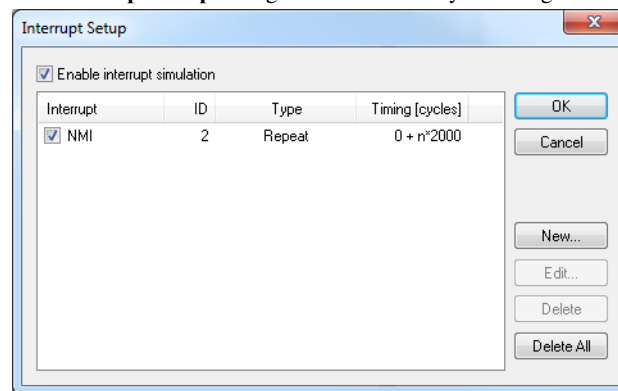
Reference information about:

- *Interrupt Setup dialog box*, page 351

- *Edit Interrupt dialog box*, page 353
- *Forced Interrupt window*, page 354
- *Interrupt Status window*, page 355
- *Interrupt Log window*, page 357
- *Interrupt Log Summary window*, page 361.

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.

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 353.

Edit

Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 353.

Delete

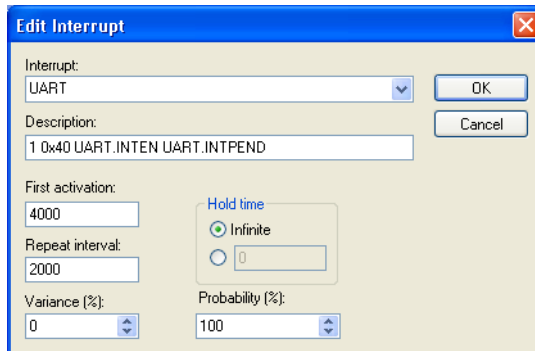
Removes the selected interrupt.

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.

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, for Cortex-M devices, populated with entries from the device description file that you have selected. For other devices, only two interrupts are available: IRQ and FIQ.

Description

A description of the selected interrupt, if available. The description is retrieved from the selected device description file and consists of a string describing the priority, vector offset, enable bit, and pending bit, separated by space characters. The enable bit and pending bit are optional. It is possible to have none, only the enable bit, or both. For interrupts specified using the system macro `__orderInterrupt`, the **Description** box is empty.

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

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

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

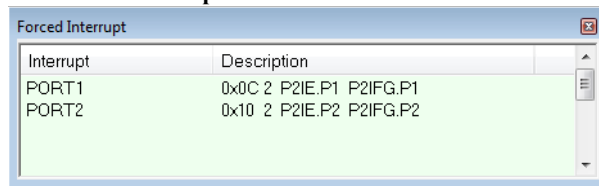
Specify how long, in cycles, the interrupt remains pending until removed if it has not been processed. If you select **Infinite**, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

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.



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 351.
- 2 Double-click the interrupt in the **Forced Interrupt** window, or activate by using the **Force** command available on the context menu.

Requirements

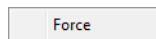
The C-SPY simulator.

Display area

This area lists all available interrupts and their definitions. This 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	Id	Type	Status	Next Time	Timing [cycles]
NMI	2	Forced	Executing	--	--
IRQ0	1	Repeat (macro)	Suspended	--	--
NMI	0	Repeat	Idle	4345	2000 + n*2345
IRQ0	1	Repeat (macro)	Idle	5020	3010 + n*2010

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	IRQT0	Triggered	0x13E8	
111.26 us	IRQT0	Enter	0x13F0	
135.78 us	IRQT1	Enter	0x1126	
148.72 us	IRQT1	Leave	0x1378	12.94 us
189.34 us	Overflow			
207.30 us	IRQT0	Leave	0x1126	96.04 us
230.00 us	IRQT0	Triggered	0x1110	
231.34 us	IRQT0	Enter	0x1126	
240.26 us	IRQT0	Leave	0x1122	8.92 us
300.00 us	IRQT1	Enter	---	
371.12 us	IRQT1	Leave	0x1120	71.12 us

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 entries in the beginning of the buffer are erased.

For more information, see *Getting started using interrupt logging*, page 350.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 224.

Requirements

One of these alternatives:

- The C-SPY simulator
- An I-jet or I-jet Trace in-circuit debugging probe or a JTAGjet debug probe, and an SWD interface between the debug probe and the target system
- A J-Link or J-Trace debug probe with an SWD interface between the debug probe and the target system
- An ST-LINK debug probe with an SWD interface between the debug probe and the target system.

Display area for the C-SPY hardware debugger drivers

This area contains these columns:

Time

The time for the interrupt entrance, based on the CPU clock frequency specified in the **SWO Configuration** dialog box.

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

This column is available when you have selected **Show Time** from the context menu. If the **Show Time** command is not available, the **Time** column is displayed by default.

Cycles

The number of cycles from the start of the execution until the event.

A cycle count displayed in italics indicates an approximative value. Italics is used when the target system has not been able to collect a correct value, but instead had to approximate it.

This column is available when you have selected **Show Cycles** from the context menu provided that the C-SPY driver you are using supports it.

Interrupt

The name of the interrupt source where the interrupt occurred. If the column displays **Overflow** in red, the communication channel failed to transmit all interrupt logs from the target system.

Status

The event status of the interrupt:

Enter, the interrupt is currently executing.

Leave, the interrupt has finished executing.

Program Counter*

The address of the interrupt handler.

Execution Time/Cycles

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

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

Display area for the C-SPY simulator

This area contains these columns:

Time

The time for the interrupt entrance, based on an internally specified clock frequency.

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

Cycles

The number of cycles from the start of the execution until the event.

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

Interrupt

The interrupt as defined in the device description file.

Status

Shows the event status of the interrupt:

Triggered, the interrupt has passed its activation time.

Forced, the same as Triggered, but the interrupt was forced from the Forced Interrupt window.

Enter, the interrupt is currently executing.

Leave, the interrupt has been executed.

Expired, the interrupt hold time has expired without the interrupt being executed.

Rejected, the interrupt has been rejected because the necessary interrupt registers were not set up to accept the interrupt.

Program Counter

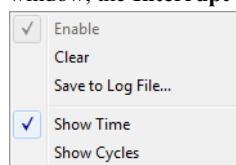
The value of the program counter when the event occurred.

Execution Time/Cycles

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

Context menu

This context menu is available in the **Data Log** window, the **Data Log Summary** window, the **Interrupt Log** window, and in the **Interrupt Log Summary** window:



Note: The commands are the same in each window, but they only operate on the specific window.

These commands are available:

Enable

Enables the logging system. The system will log information also when the window is closed.

Clear

Deletes the log information. Note that this will happen also when you reset the debugger.

Save to log file

Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF`. An **X** in the **Approx** column indicates that the timestamp is an approximation.

Show Time

Displays the **Time** column in the **Data Log** window and in the **Interrupt Log** window, respectively.

This menu command might not be available in the C-SPY driver you are using, which means that the **Time** column is by default displayed in the **Data Log** window.

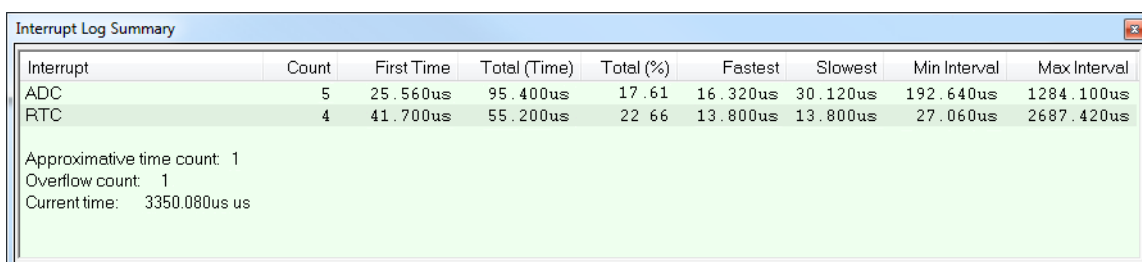
Show Cycles

Displays the **Cycles** column in the **Data Log** window and in the **Interrupt Log** window, respectively.

This menu command might not be available in the C-SPY driver you are using, which means that the **Cycles** column is not supported.

Interrupt Log Summary window

The Interrupt Log Summary window is available from the C-SPY driver menu.



Interrupt	Count	First Time	Total (Time)	Total (%)	Fastest	Slowest	Min Interval	Max Interval
ADC	5	25.560us	95.400us	17.61	16.320us	30.120us	192.640us	1284.100us
RTC	4	41.700us	55.200us	22.66	13.800us	13.800us	27.060us	2687.420us

Approximative time count: 1
 Overflow count: 1
 Current time: 3350.080us us

This window displays a summary of logs of entrances to and exits from interrupts.

For more information, see *Getting started using interrupt logging*, page 350.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 224.

Requirements

One of these alternatives:

- The C-SPY simulator
- An I-jet or I-jet Trace in-circuit debugging probe or a JTAGjet debug probe, and an SWD interface between the debug probe and the target system
- A J-Link or J-Trace debug probe with an SWD interface between the debug probe and the target system
- An ST-LINK debug probe with an SWD interface between the debug probe and the target system.

Display area for the C-SPY simulator

Each row in this area displays statistics about the specific interrupt based on the log information in these columns:

Interrupt

The type of interrupt that occurred.

At the bottom of the column, the current time or cycles is displayed—the number of cycles or the execution time since the start of execution. Overflow count and approximative time count is always zero.

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

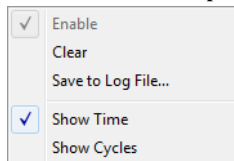
The longest time between two interrupts of this type.

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 in the **Data Log** window, the **Data Log Summary** window, the **Interrupt Log** window, and in the **Interrupt Log Summary** window:



Note: The commands are the same in each window, but they only operate on the specific window.

These commands are available:

Enable

Enables the logging system. The system will log information also when the window is closed.

Clear

Deletes the log information. Note that this will happen also when you reset the debugger.

Save to log file

Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF`. An **X** in the **Approx** column indicates that the timestamp is an approximation.

Show Time

Displays the **Time** column in the **Data Log** window and in the **Interrupt Log** window, respectively.

This menu command might not be available in the C-SPY driver you are using, which means that the **Time** column is by default displayed in the **Data Log** window.

Show Cycles

Displays the **Cycles** column in the **Data Log** window and in the **Interrupt Log** window, respectively.

This menu command might not be available in the C-SPY driver you are using, which means that the **Cycles** column is not supported.

C-SPY macros

- Introduction to C-SPY macros
- Using C-SPY macros
- Reference information on the macro language
- Reference information on reserved setup macro function names
- Reference information on C-SPY system macros
- Graphical environment for macros

Introduction to C-SPY macros

These topics are covered:

- Reasons for using C-SPY macros
- Briefly about using C-SPY macros
- Briefly about setup macro functions and files
- Briefly about the macro language

REASONS FOR USING C-SPY MACROS

You can use C-SPY macros either by themselves or in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. Some examples where macros can be useful:

- Automating the debug session, for instance with trace printouts, printing values of variables, and setting breakpoints.
- Hardware configuring, such as initializing hardware registers.
- Feeding your application with simulated data during runtime.
- Simulating peripheral devices, see the chapter *Interrupts*. This only applies if you are using the simulator driver.
- Developing small debug utility functions, for instance calculating the stack depth, see the provided example `stack.mac` located in the directory `\arm\src\`.

BRIEFLY ABOUT USING C-SPY MACROS

To use C-SPY macros, you should:

- Write your macro variables and functions and collect them in one or several *macro files*
- Register your macros
- Execute your macros.

For registering and executing macros, there are several methods to choose between. Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to register or execute your macro.

BRIEFLY ABOUT SETUP MACRO FUNCTIONS AND FILES

There are some reserved *setup macro function names* that you can use for defining macro functions which will be called at specific times, such as:

- Once after communication with the target system has been established but before downloading the application software
- Once after your application software has been downloaded
- Each time the reset command is issued
- Once when the debug session ends.

To define a macro function to be called at a specific stage, you should define and register a macro function with one of the reserved names. For instance, if you want to clear a specific memory area before you load your application software, the macro setup function `execUserPreload` should be used. This function is also suitable if you want to initialize some CPU registers or memory-mapped peripheral units before you load your application software.

You should define these functions in a *setup macro file*, which you can load before C-SPY starts. Your macro functions will then be automatically registered each time you start C-SPY. This is convenient if you want to automate the initialization of C-SPY, or if you want to register multiple setup macros.

For more information about each setup macro function, see *Reference information on reserved setup macro function names*, page 377.

Remapping memory

A common feature of many ARM-based processors is the ability to remap memory. After a reset, the memory controller typically maps address zero to non-volatile memory, such as flash. By configuring the memory controller, the system memory can be remapped to place RAM at zero and non-volatile memory higher up in the address map. By doing this, the exception table will reside in RAM and can be easily modified

when you download code to the target hardware. To handle this in C-SPY, the setup macro function `execUserPreload()` is suitable. For an example, see *Remapping memory*, page 56.

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 372.

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 55.

REGISTERING C-SPY MACROS—AN OVERVIEW

C-SPY must know that you intend to use your defined macro functions, and thus you must *register* your macros. There are various ways to register macro functions:

- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files*, page 369.
- You can register macros interactively in the **Macro Registration** window, see *Macro Registration window*, page 433. Registered macros appear in the **Debugger Macros** window, see *Debugger Macros window*, page 435.
- 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 413.

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 369.
- 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 370.
- The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is more specified on designed for C-SPY macros. See *Macro Quicklaunch window*, page 437.

- 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 370.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to execute your macro.

REGISTERING AND EXECUTING USING SETUP MACROS AND SETUP FILES

It can be convenient to register a macro file during the C-SPY startup sequence. To do this, specify a macro file which you load before starting the debug session. Your macro functions will be automatically registered each time you start the debugger.

If you use the reserved setup macro function names to define the macro functions, you can define exactly at which stage you want the macro function to be executed.

To define a setup macro function and load it during C-SPY startup:

- 1 Create a new text file where you can define your macro function.

For example:

```
execUserSetup()
{
    ...
    __registerMacroFile("MyMacroUtils.mac");
    __registerMacroFile("MyDeviceSimulation.mac");
}
```

This macro function registers the additional macro files `MyMacroUtils.mac` and `MyDeviceSimulation.mac`. Because the macro function is defined with the function name `execUserSetup`, it will be executed directly after your application has been downloaded.

- 2 Save the file using the filename extension `mac`.
- 3 Before you start C-SPY, choose **Project>Options>Debugger>Setup**. Select **Use Setup file** and choose the macro file you just created.

The macros will now be registered during the C-SPY startup sequence.

EXECUTING MACROS USING QUICK WATCH

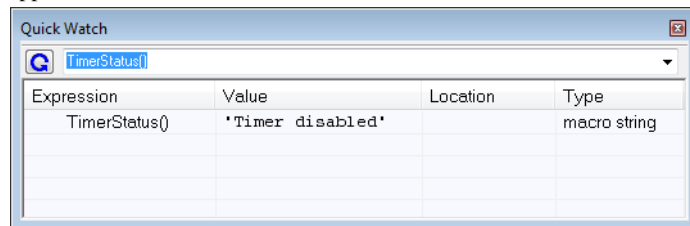
The **Quick Watch** window lets you dynamically choose when to execute a macro function.

- 1 Consider this simple macro function that checks the status of a timer enable bit:

```
TimerStatus()
{
    if ((TimerStatreg & 0x01) != 0) /* Checks the status of reg */
        return "Timer enabled"; /* C-SPY macro string used */
    else
        return "Timer disabled"; /* C-SPY macro string used */
}
```

- 2 Save the macro function using the filename extension `mac`.
- 3 To load the macro file, choose **View>Macros>Macro Registration**. The **Macro Registration** window is displayed. Click **Add** and locate the file using the file browser. The macro file appears in the list of macros in the **Macro Registration** window.
- 4 Select the macro you want to register and your macro will appear in the **Debugger Macros** window.
- 5 Choose **View>Quick Watch** to open the **Quick Watch** window, type the macro call `TimerStatus()` in the text field and press Return,

Alternatively, in the macro file editor window, select the macro function name `TimerStatus()`. Right-click, and choose **Quick Watch** from the context menu that appears.



The macro will automatically be displayed in the **Quick Watch** window.

For more information, see *Quick Watch window*, page 112.

EXECUTING A MACRO BY CONNECTING IT TO A BREAKPOINT

You can connect a macro to a breakpoint. The macro will then be executed when the breakpoint is triggered. The advantage is that you can stop the execution at locations of particular interest and perform specific actions there.



For instance, you can easily produce log reports containing information such as how the values of variables, symbols, or registers change. To do this you might set a breakpoint on a suspicious location and connect a log macro to the breakpoint. After the execution you can study how the values of the registers have changed.

To create a log macro and connect it to a breakpoint:

- 1 Assume this skeleton of a C function in your application source code:

```
int fact(int x)
{
    ...
}
```

- 2 Create a simple log macro function like this example:

```
logfact()
{
    __message "fact(" ,x, " )";
}
```

The `__message` statement will log messages to the Log window.

Save the macro function in a macro file, with the filename extension `mac`.

- 3 To register the macro, choose **View>Macros>Macro Registration** to open the **Macro Registration** window and add your macro file to the list. Select the file to register it. Your macro function will appear in the **Debugger Macros** window.
- 4 To set a code breakpoint, click the **Toggle Breakpoint** button on the first statement within the function `fact` in your application source code. Choose **View>Breakpoints** to open the **Breakpoints** window. Select your breakpoint in the list of breakpoints and choose the **Edit** command from the context menu.
- 5 To connect the log macro function to the breakpoint, type the name of the macro function, `logfact()`, in the **Action** field and click **Apply**. Close the dialog box.
- 6 Execute your application source code. When the breakpoint is triggered, the macro function will be executed. You can see the result in the **Log** window.
 - Note that the expression in the **Action** field is evaluated only when the breakpoint causes the execution to really stop. If you want to log a value and then automatically continue execution, you can either:
 - Use a **Log** breakpoint, see *Log breakpoints dialog box*, page 145
 - Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 136.
- 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 375.

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 372
- *Macro variables*, page 373
- *Macro parameters*, page 373
- *Macro strings*, page 374
- *Macro statements*, page 374
- *Formatted output*, page 375.

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 92.

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>	myvar is now type double, value 3.5.
<code>myvar = (int*)i;</code>	myvar is now type pointer to int, and the value is the same as i.

Table 16: 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 477.

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 375.

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 92.

Conditional statements

```
if (expression)
    statement
```

```

if (expression)
    statement
else
    statement

```

Loop statements

```

for (init_expression; cond_expression; update_expression)
    statement

```

```

while (expression)
    statement

```

```

do
    statement
while (expression);

```

Return statements

```
return;
```

```
return expression;
```

If the return value is not explicitly set, signed int 0 is returned by default.

Blocks

Statements can be grouped in blocks.

```

{
    statement1
    statement2
    .
    .
    .
    statementN
}

```

FORMATTED OUTPUT

C-SPY provides various methods for producing formatted output:

<code>__message <i>argList</i>;</code>	Prints the output to the Debug Log window.
<code>__fmessage <i>file</i>, <i>argList</i>;</code>	Prints the output to the designated file.
<code>__smessage <i>argList</i>;</code>	Returns a string containing the formatted output.

where *argList* is a comma-separated list of C-SPY expressions or strings, and *file* is the result of the `__openFile` system macro, see `__openFile`, page 407.

To produce messages in the **Debug Log** window:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2,
" in the Log window.";
```

This produces this message in the Log window:

```
This line prints the values 42 and 37 in the Log window.
```

To write the output to a designated file:

```
__fmessage myfile, "Result is ", res, "!\n";
```

To produce strings:

```
myMacroVar = __smmessage 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 366.

Reference information about:

- `execUserPreload`
- `execUserExecutionStarted`
- `execUserExecutionStopped`
- `execUserFlashInit`
- `execUserSetup`
- `execUserFlashReset`
- `execUserPreReset`
- `execUserReset`
- `execUserExit`
- `execUserFlashExit`

execUserPreload

Syntax	<code>execUserPreload</code>
For use with	All C-SPY drivers.
Description	Called after communication with the target system is established but before downloading the target application.

Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.

execUserExecutionStarted

Syntax	<code>execUserExecutionStarted</code>
For use with	All C-SPY drivers.
Description	Called when the debugger is about to start or resume execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window.

execUserExecutionStopped


Syntax	<code>execUserExecutionStopped</code>
For use with	All C-SPY drivers.
Description	Called when the debugger has stopped execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window.

execUserFlashInit

Syntax	<code>execUserFlashInit</code>
For use with	The C-SPY hardware debugger drivers.
Description	Called once before the flash loader is downloaded to RAM. Implement this macro typically for setting up the memory map required by the flash loader. This macro is only called when you are programming flash, and it should only be used for flash loader functionality.

execUserSetup

Syntax	<code>execUserSetup</code>
For use with	All C-SPY drivers.

Description	<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> <p> If you define interrupts or breakpoints in a macro file that is executed at system start (using <code>execUserSetup</code>) we strongly recommend that you also make sure that they are removed at system shutdown (using <code>execUserExit</code>). An example is available in <code>SetupSimple.mac</code>, see the tutorials in the Information Center.</p> <p>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 <code>execUserSetup</code> is executed again. This seriously affects the execution speed.</p>
-------------	--

execUserFlashReset

Syntax	<code>execUserFlashReset</code>
For use with	The C-SPY hardware debugger drivers.
Description	Called once after the flash loader is downloaded to RAM, but before execution of the flash loader. This macro is only called when you are programming flash, and it should only be used for flash loader functionality.

execUserPreReset

Syntax	<code>execUserPreReset</code>
For use with	All C-SPY drivers.
Description	Called each time just before the reset command is issued. Implement this macro to set up any required device state.

execUserReset

Syntax	<code>execUserReset</code>
For use with	All C-SPY drivers.
Description	Called each time just after the reset command is issued. Implement this macro to set up and restore data.

execUserExit

Syntax	<code>execUserExit</code>
For use with	All C-SPY drivers.
Description	Called once when the debug session ends. Implement this macro to save status data etc.

execUserFlashExit

Syntax	<code>execUserFlashExit</code>
For use with	The C-SPY hardware debugger drivers.
Description	Called once when the flash programming ends. Implement this macro to save status data etc. This macro is useful for flash loader functionality.

Reference information on C-SPY system macros

This section gives reference information about each of the C-SPY system macros.

This table summarizes the pre-defined system macros:

Macro	Description
<code>__cancelAllInterrupts</code>	Cancels all ordered interrupts
<code>__cancelInterrupt</code>	Cancels an interrupt
<code>__clearBreak</code>	Clears a breakpoint
<code>__closeFile</code>	Closes a file that was opened by <code>__openFile</code>
<code>__delay</code>	Delays execution
<code>__disableInterrupts</code>	Disables generation of interrupts
<code>__driverType</code>	Verifies the driver type
<code>__emulatorSpeed</code>	Sets the emulator clock frequency
<code>__emulatorStatusCheckOnRead</code>	Enables or disables the verification of the CPSR register after each read operation
<code>__enableInterrupts</code>	Enables generation of interrupts

Table 17: Summary of system macros

Macro	Description
<code>__evaluate</code>	Interprets the input string as an expression and evaluates it.
<code>__fillMemory8</code>	Fills a specified memory area with a byte value.
<code>__fillMemory16</code>	Fills a specified memory area with a 2-byte value.
<code>__fillMemory32</code>	Fills a specified memory area with a 4-byte value.
<code>__gdbserver_exec_command</code>	Send strings or commands to the GDB Server
<code>__getSelectedCore</code>	Gets the number of the current core.
<code>__getTracePortSize</code>	Returns the width of the trace port
<code>__hasDAPRegs</code>	Returns true if the C-SPY driver supports the macros <code>__readAPReg</code> , <code>__readDPRReg</code> , and <code>__writeAPReg</code> , and <code>__writeDPRReg</code> .
<code>__hwJetResetWithStrategy</code>	Performs a hardware reset and a halt of the target CPU
<code>__hwReset</code>	Performs a hardware reset and a halt of the target CPU
<code>__hwResetRunToBp</code>	Performs a hardware reset and then executes to the specified address
<code>__hwResetWithStrategy</code>	Performs a hardware reset and halt with delay of the target CPU
<code>__isBatchMode</code>	Checks if C-SPY is running in batch mode or not.
<code>__jlinkExecCommand</code>	Sends a low-level command to the J-Link/J-Trace driver
<code>__jtagCommand</code>	Sends a low-level command to the JTAG instruction register
<code>__jtagCP15IsPresent</code>	Checks if coprocessor CP15 is available
<code>__jtagCP15ReadReg</code>	Returns the coprocessor CP15 register value
<code>__jtagCP15WriteReg</code>	Writes to the coprocessor CP15 register
<code>__jtagData</code>	Sends a low-level data value to the JTAG data register
<code>__jtagRawRead</code>	Returns the read data from the JTAG interface
<code>__jtagRawSync</code>	Writes accumulated data to the JTAG interface
<code>__jtagRawWrite</code>	Accumulates data to be transferred to the JTAG
<code>__jtagResetTRST</code>	Resets the ARM TAP controller via the TRST JTAG signal

Table 17: Summary of system macros (Continued)

Macro	Description
<code>__loadImage</code>	Loads an image.
<code>__memoryRestore</code>	Restores the contents of a file to a specified memory zone
<code>__memorySave</code>	Saves the contents of a specified memory area to a file
<code>__messageBoxYesCancel</code>	Displays a Yes/Cancel dialog box for user interaction
<code>__messageBoxYesNo</code>	Displays a Yes/No dialog box for user interaction
<code>__openFile</code>	Opens a file for I/O operations
<code>__orderInterrupt</code>	Generates an interrupt
<code>__popSimulatorInterruptExecutingStack</code>	Informs the interrupt simulation system that an interrupt handler has finished executing
<code>__readAPReg</code>	Reads from an AP register
<code>__readDPReg</code>	Reads from a DP register
<code>__readFile</code>	Reads from the specified file
<code>__readFileByte</code>	Reads one byte from the specified file
<code>__readMemory8,</code> <code>__readMemoryByte</code>	Reads one byte from the specified memory location
<code>__readMemory16</code>	Reads two bytes from the specified memory location
<code>__readMemory32</code>	Reads four bytes from the specified memory location
<code>__registerMacroFile</code>	Registers macros from the specified file
<code>__resetFile</code>	Rewinds a file opened by <code>__openFile</code>
<code>__restoreSoftwareBreakpoints</code>	Restores any breakpoints that were destroyed during system startup.
<code>__selectCore</code>	Switches focus from the current core to the specified core.
<code>__setCodeBreak</code>	Sets a code breakpoint
<code>__setDataBreak</code>	Sets a data breakpoint
<code>__setDataLogBreak</code>	Sets a data log breakpoint
<code>__setLogBreak</code>	Sets a log breakpoint
<code>__setSimBreak</code>	Sets a simulation breakpoint
<code>__setTraceStartBreak</code>	Sets a trace start breakpoint
<code>__setTraceStopBreak</code>	Sets a trace stop breakpoint

Table 17: Summary of system macros (Continued)

Macro	Description
<code>__sourcePosition</code>	Returns the file name and source location if the current execution location corresponds to a source location
<code>__strFind</code>	Searches a given string for the occurrence of another string
<code>__subString</code>	Extracts a substring from another string
<code>__targetDebuggerVersion</code>	Returns the version of the target debugger
<code>__toLowerCase</code>	Returns a copy of the parameter string where all the characters have been converted to lower case
<code>__toString</code>	Prints strings
<code>__toUpperCase</code>	Returns a copy of the parameter string where all the characters have been converted to upper case
<code>__unloadImage</code>	Unloads a debug image
<code>__writeAPReg</code>	Writes to an AP register
<code>__writeDPRreg</code>	Writes to a DP register
<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 17: Summary of system macros (Continued)

__cancelAllInterrupts

Syntax	<code>__cancelAllInterrupts()</code>
Return value	<code>int 0</code>
For use with	The C-SPY Simulator.
Description	Cancels all ordered interrupts.

__cancelInterrupt

Syntax `__cancelInterrupt(interrupt_id)`

Parameters *interrupt_id*

The value returned by the corresponding `__orderInterrupt` macro call (unsigned long).

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 18: `__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 125.

__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 `__delay(value)`

Parameters *value*
The number of milliseconds to delay execution.

Return value `int 0`

For use with All C-SPY drivers.

Description Delays execution the specified number of milliseconds.

`__disableInterrupts`

Syntax `__disableInterrupts()`

Return value

Result	Value
Successful	<code>int 0</code>
Unsuccessful	Non-zero error number

Table 19: `__disableInterrupts` return values

For use with The C-SPY Simulator.

Description Disables the generation of interrupts.

`__driverType`

Syntax `__driverType(driver_id)`

Parameters

driver_id

A string corresponding to the driver you want to check for. Choose one of these:

"sim" corresponds to the simulator driver.

"angel" corresponds to the C-SPY Angel driver

"cmsisdap" corresponds to the C-SPY CMSIS-DAP driver
 "gdbserv" corresponds to the C-SPY GDB Server driver
 "generic" corresponds to third-party drivers
 "ijet" corresponds to the C-SPY I-jet/JTAGjet driver
 "jlink" corresponds to the C-SPY J-Link/J-Trace driver
 "jtag" corresponds to the C-SPY Macraigor driver
 "lmiftdi" corresponds to the C-SPY TI Stellaris driver
 "xds" corresponds to the C-SPY TI XDS driver
 "rdi" corresponds to the C-SPY RDI driver
 "rom" corresponds to the C-SPY IAR ROM-monitor driver
 "stlink" corresponds to the C-SPY ST-LINK driver.

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 20: `__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.

`__emulatorSpeed`

Syntax

`__emulatorSpeed(speed)`

Parameters

speed

The emulator speed in Hz. Use 0 (zero) to make the speed automatically detected. Use -1 for adaptive speed (only for emulators supporting adaptive speed).

Return value

Result	Value
Successful	The previous speed, or 0 (zero) if unknown
Unsuccessful; the speed is not supported by the emulator	-1

Table 21: `__emulatorSpeed` return values

For use with

The C-SPY hardware drivers

Description

Sets the emulator clock frequency. For JTAG interfaces, this is the JTAG clock frequency as seen on the TCK signal.

Example

```
__emulatorSpeed(0)
```

Sets the emulator speed to be automatically detected.

`__emulatorStatusCheckOnRead`

Syntax

```
__emulatorStatusCheckOnRead(status)
```

Parameters

status Use 0 to enable checks (default). Use 1 to disable checks.

Return value

int 0

For use with

The C-SPY J-Link/J-Trace driver

For the C-SPY I-jet/JTAG-jet driver, this macro is recognized, but has no effect.

Description

Enables or disables the driver verification of CPSR (current processor status register) after each read operation. Typically, this macro can be used for initiating JTAG connections on some CPUs, like Texas Instruments' TMS470R1B1M.

Note: Enabling this verification can cause problems with some CPUs, for example if invalid CPSR values are returned. However, if this verification is disabled (`SetCheckModeAfterRead = 0`), the success of read operations cannot be verified and possible data aborts are not detected.

Example

```
__emulatorStatusCheckOnRead(1)
```

Disables the checks for data aborts on memory reads.

__enableInterrupts

Syntax `__enableInterrupts()`

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 22: `__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 23: `__evaluate` return values

For use with All C-SPY drivers.

Description This macro interprets the input string as an expression and evaluates it. The result is stored in a variable pointed to by *valuePtr*.

Example

This example assumes that the variable `i` is defined and has the value 5:

```
__evaluate("i + 3", &myVar)
```

The macro variable `myVar` is assigned the value 8.

__fillMemory8

Syntax	<code>__fillMemory8(value, address, zone, length, format)</code>								
Parameters	<p><i>value</i> An integer that specifies the value.</p> <p><i>address</i> An integer that specifies the memory start address.</p> <p><i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 163.</p> <p><i>length</i> An integer that specifies how many bytes are affected.</p> <p><i>format</i> One of these alternatives:</p> <table> <tr> <td>Copy</td> <td><i>value</i> will be copied to the specified memory area.</td> </tr> <tr> <td>AND</td> <td>An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> <tr> <td>OR</td> <td>An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> <tr> <td>XOR</td> <td>An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> </table>	Copy	<i>value</i> will be copied to the specified memory area.	AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.	OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.	XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
Copy	<i>value</i> will be copied to the specified memory area.								
AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
Return value	<code>int 0</code>								
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(value, address, zone, length, format)</code>
Parameters	<p><i>value</i> An integer that specifies the value.</p>

address

An integer that specifies the memory start address.

zone

A string that specifies the memory zone, see *C-SPY memory zones*, page 163.

length

An integer that defines how many 2-byte entities to be affected.

format

One of these alternatives:

Copy	<i>value</i> will be copied to the specified memory area.
AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.

Return value	int 0
For use with	All C-SPY drivers.
Description	Fills a specified memory area with a 2-byte value.
Example	<pre>__fillMemory16(0xCDCD, 0x7000, "", 0x200, "Copy");</pre>

__fillMemory32

Syntax `__fillMemory32(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 163.

length

An integer that defines how many 4-byte entities to be affected.

format

One of these alternatives:

Copy	<i>value</i> will be copied to the specified memory area.
AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.

Return value	int 0
For use with	All C-SPY drivers.
Description	Fills a specified memory area with a 4-byte value.
Example	<code>__fillMemory32(0x0000FFFF, 0x4000, "", 0x1000, "XOR");</code>

__gdbserver_exec_command

Syntax	<code>__gdbserver_exec_command("string")</code>
Parameters	"string" String or command sent to the GDB Server; see its documentation for more information.
Applicability	The C-SPY C-SPY GDB Server driver
Description	Use this option to send strings or commands to the GDB Server.

__getSelectedCore

Syntax `__getSelectedCore()`

Return value The current core. The cores are numbered from 0 and upwards.

For use with The C-SPY simulator.
The C-SPY I-jet/JTAGjet driver.

Description Gets the number of the current core.

Example

```
test ()
{
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x,
  "\n";
  __selectCore(0);
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x,
  "\n";
  __selectCore(1);
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x,
  "\n";
}
```

A typical result of the above macro would be (assuming that the original core was number 1):

```
Core: 1 pc = 0000213C
Core: 0 pc = 00000494
Core: 1 pc = 0000213C
```

See also `__selectCore`, page 415.

__getTracePortSize

Syntax `__getTracePortSize`

Result	Value
The width of the trace port in bits.	1, 2, 4, 8, or 16.

Table 24: `__getTracePortSize` return values

For use with The C-SPY I-jet/JTAGjet driver
The C-SPY J-Link/J-Trace driver

Description Returns the width of the trace port.

See also *ETM Trace Settings dialog box*, page 208 and *ETM Trace Settings dialog box (J-Link/J-Trace)*, page 210, respectively.

__hasDAPRegs

Syntax `__hasDAPRegs()`

Return value

Result	Value
The C-SPY driver supports the macros <code>__readAPReg</code> , <code>__readDPReg</code> , <code>__writeAPReg</code> , and <code>__writeDPReg</code> for the current CPU core.	true
The C-SPY driver does not support the macros <code>__readAPReg</code> , <code>__readDPReg</code> , <code>__writeAPReg</code> , and <code>__writeDPReg</code> for the current CPU core.	false

Table 25: `__hasDAPRegs` return values

For use with The C-SPY hardware drivers

Description This macro returns true if the C-SPY driver supports the macros `__readAPReg`, `__readDPReg`, `__writeAPReg`, and `__writeDPReg` for the current CPU core, otherwise it returns false.

__hwJetResetWithStrategy

Syntax `__hwJetResetWithStrategy(halt_delay, strategy)`

Parameters

<i>halt_delay</i>	The delay, in milliseconds, between the end of the reset pulse and the halt of the CPU. Use 0 (zero) to make the CPU halt immediately after reset; only when <i>strategy</i> is set to 0.
<i>strategy</i>	The reset strategy number. For information about supported reset strategies, see <i>--jet_standard_reset</i> , page 470.

Return value

Result	Value
Successful. The delay feature is not supported by the debugging probe	-1

Table 26: `__hwJetResetWithStrategy` return values

Result	Value
Unsuccessful. The reset strategy is not supported by the debugging probe	-3
Unsuccessful. Other	-4

Table 26: `__hwJetResetWithStrategy` return values (Continued)

For use with	The C-SPY I-jet/JTAGjet driver
Description	Specifies the reset strategy to perform.
Example	<code>__hwJetResetWithStrategy(0, 2)</code> Performs a hardware reset.

`__hwReset`

Syntax	<code>__hwReset(halt_delay)</code>
Parameters	<i>halt_delay</i> The delay, in milliseconds, between the end of the reset pulse and the halt of the CPU. Use 0 (zero) to make the CPU halt immediately after reset

Return value

Result	Value
Successful. The actual delay value implemented by the emulator	≥ 0
Successful. The delay feature is not supported by the emulator	-1
Unsuccessful. Hardware reset is not supported by the emulator	-2

Table 27: `__hwReset` return values

For use with	This system macro is available for all JTAG interfaces.
Description	Performs a hardware reset and halt of the target CPU.
Example	<code>__hwReset(0)</code> Resets the CPU and immediately halts it.

`__hwResetRunToBp`

Syntax	<code>__hwResetRunToBp(strategy, breakpoint_address, timeout)</code>
--------	--

Parameters

<i>strategy</i>	For information about supported reset strategies in the C-SPY I-jet/JTAG-jet driver, see <i>--jet_standard_reset</i> , page 470. For information about supported reset strategies in the C-SPY J-Link driver, see the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> .
<i>breakpoint_address</i>	The address of the breakpoint to execute to, specified as an integer value (symbols cannot be used).
<i>timeout</i>	A time out for the breakpoint, specified in milliseconds. If the breakpoint is not reached within the specified time, the core will be halted.

Return value

Value	Result
>=0	Successful. The approximate execution time in ms until the breakpoint is hit.
-2	Unsuccessful. Hardware reset is not supported by the emulator.
-3	Unsuccessful. The reset strategy is not supported by the emulator.

Table 28: *__hwResetRunToBp* return values

For use with

The C-SPY CMSIS-DAP driver
 The C-SPY I-jet/JTAGjet driver
 The C-SPY J-Link/J-Trace driver

Description

Performs a hardware reset, sets a breakpoint at the specified address, executes to the breakpoint, and then removes it. The breakpoint address should be the start address of the downloaded image after it has been copied to RAM.

This macro is intended for running a boot loader that copies the application image from flash to RAM. The macro should be executed after the image has been downloaded to flash, but before the image is verified. The macro can be run in `execUserFlashExit` or `execUserPreload`.

Example

```
__hwResetRunToBp(0, 0x400000, 10000)
```

Resets the CPU with the reset strategy 0 and executes to the address 0x400000. If the breakpoint is not reached within 10 seconds, execution stops in accordance with the specified time out.

__hwResetWithStrategy

Syntax `__hwResetWithStrategy(halt_delay, strategy)`

Parameters

halt_delay The delay, in milliseconds, between the end of the reset pulse and the halt of the CPU. Use 0 (zero) to make the CPU halt immediately after reset; only when *strategy* is set to 0.

strategy The C-SPY I-jet/JTAGjet driver only supports strategy 2 (hardware reset). For information about supported reset strategies in the C-SPY J-Link driver, see the *IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores*.

Return value

Result	Value
Successful. The actual delay in milliseconds, as implemented by the emulator	>=0
Successful. The delay feature is not supported by the emulator	-1
Unsuccessful. Hardware reset is not supported by the emulator	-2
Unsuccessful. The reset strategy is not supported by the emulator	-3

Table 29: __hwResetWithStrategy return values

For use with

The C-SPY I-jet/JTAGjet driver

The C-SPY J-Link/J-Trace driver

This macro exists also in the other C-SPY hardware drivers, but there it has no effect.

Description

Performs a hardware reset and a halt with delay of the target CPU.

Example

```
__hwResetWithStrategy(0,1)
```

Resets the CPU and halts it using a breakpoint at memory address zero.

__hwRunToBreakpoint

Syntax `__hwRunToBreakpoint(breakpoint_address, timeout)`

Parameters

breakpoint_address The address of the breakpoint to execute to, specified as an integer value (symbols cannot be used).

timeout A time out for the breakpoint, specified in milliseconds. If the breakpoint is not reached within the specified time, the core will be halted.

Return value

Value	Result
>=0	Successful. The approximate execution time in ms until the breakpoint is hit.
-1	Failed to set the breakpoint.
-2	Failed to stop at the breakpoint before timeout.

Table 30: *__hwRunToBreakpoint* return values

For use with

The C-SPY CMSIS-DAP driver
 The C-SPY I-jet/JTAGjet driver
 The C-SPY J-Link/J-Trace driver
 The C-SPY PE micro driver
 The C-SPY ST-LINK driver
 The C-SPY TI XDS driver.

Description

Use this macro to set a temporary breakpoint and then start the execution. When the breakpoint is triggered, the execution stops. This macro can be used for running initialization code on the target system.

Example

```
__hwRunToBreakpoint(0x20000048, 1000)
```

Sets a temporary breakpoint at the address 0x20000048, starts executing, and executes until the breakpoint is triggered or until 1000 ms have passed.

__isBatchMode

Syntax

```
__isBatchMode()
```

Return value

Result	Value
True	int 1
False	int 0

Table 31: *__isBatchMode* return values

For use with

All C-SPY drivers.

Description This macro returns True if the debugger is running in batch mode, otherwise it returns False.

__jlinkExecCommand

Syntax `__jlinkExecCommand(cmdstr)`

Parameters *cmdstr* J-Link/J-Trace command string

Return value `int 0`

For use with The C-SPY J-Link/J-Trace driver

Description Sends a low-level command to the J-Link/J-Trace driver. For a list of possible commands, see the *IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores*.

Example See the *IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores*.

See also `--jlink_exec_command`, page 474

__jtagCommand

Syntax `__jtagCommand(ir)`

Parameters *ir* can be one of:

2	SCAN_N
4	RESTART
12	INTEST
14	IDCODE
15	BYPASS

Return value `int 0`

For use with The C-SPY J-Link/J-Trace driver

Description Sends a low-level command to the JTAG instruction register `IR`.

Example

```
__jtagCommand(14);
Id = __jtagData(0,32);
```

Returns the JTAG ID of the ARM target device.

__jtagCP15IsPresent

Syntax `__jtagCP15IsPresent()`

Return value 1 if CP15 is available, otherwise 0.

For use with The C-SPY I-jet/JTAGjet driver
The C-SPY J-Link/J-Trace driver

Description Checks if the coprocessor CP15 is available.

__jtagCP15ReadReg

Syntax `__jtagCP15ReadReg(CRn, CRm, op1, op2)`

ParametersParameter The parameters—registers and operands—of the MRC instruction. For details, see the *ARM Architecture Reference Manual*. Note that `op1` should always be 0.

Return value The register value.

For use with The C-SPY I-jet/JTAGjet driver
The C-SPY J-Link/J-Trace driver

Description Reads the value of the CP15 register and returns its value.

__jtagCP15WriteReg

Syntax `__jtagCP15WriteReg(CRn, CRm, op1, op2, value)`

Parameters The parameters—registers and operands—of the MCR instruction. For details, see the *ARM Architecture Reference Manual*. Note that `op1` should always be 0. `value` is the value to be written.

Applicability	The C-SPY I-jet/JTAGjet driver The C-SPY J-Link/J-Trace driver
Description	Writes a value to the CP15 register.

__jtagData

Syntax	<code>__jtagData(<i>dr</i>, <i>bits</i>)</code>				
Parameters	<table> <tr> <td><i>dr</i></td> <td>32-bit data register value</td> </tr> <tr> <td><i>bits</i></td> <td>Number of valid bits in <i>dr</i>, both for the macro parameter and the return value; starting with the least significant bit (1...32)</td> </tr> </table>	<i>dr</i>	32-bit data register value	<i>bits</i>	Number of valid bits in <i>dr</i> , both for the macro parameter and the return value; starting with the least significant bit (1...32)
<i>dr</i>	32-bit data register value				
<i>bits</i>	Number of valid bits in <i>dr</i> , both for the macro parameter and the return value; starting with the least significant bit (1...32)				
Return value	Returns the result of the operation; the number of bits in the result is given by the <i>bits</i> parameter.				
For use with	The C-SPY J-Link/J-Trace driver				
Description	Sends a low-level data value to the JTAG data register DR. The bit shifted out of DR is returned.				
Example	<pre>__jtagCommand(14); Id = __jtagData(0, 32);</pre> <p>Returns the JTAG ID of the ARM target device.</p>				

__jtagRawRead

Syntax	<code>__jtagRawRead(<i>bitpos</i>, <i>numbits</i>)</code>				
Parameters	<table> <tr> <td><i>bitpos</i></td> <td>The start bit position in the returned JTAG bits to return data from</td> </tr> <tr> <td><i>numbits</i></td> <td>The number of bits to read. The maximum value is 32.</td> </tr> </table>	<i>bitpos</i>	The start bit position in the returned JTAG bits to return data from	<i>numbits</i>	The number of bits to read. The maximum value is 32.
<i>bitpos</i>	The start bit position in the returned JTAG bits to return data from				
<i>numbits</i>	The number of bits to read. The maximum value is 32.				
Applicability	The J-Link/J-Trace driver				

Description Returns the data read from the JTAG TDO. Only the least significant bits contain data; the last bit read is from the least significant bit. This function can be called an arbitrary number of times to get all bits returned by an operation. This function also makes an implicit synchronization of any accumulated write bits.

Example The following piece of pseudocode illustrates how the data is written to the JTAG (on the TMS and TDI pins) and read (from TDO):

```
__var Id;
__var BitPos;
/*****
 *
 * ReadId()
 */
ReadId() {
__message "Reading JTAG Id\n";
__jtagRawWrite(0, 0x1f, 6); /* Goto IDLE via RESET state */
__jtagRawWrite(0, 0x1, 3); /* Enter DR scan chain */
BitPos = __jtagRawWrite(0, 0x80000000, 32); /* Shift 32 bits
      into DR. Remember BitPos for Read operation */
__jtagRawWrite(0, 0x1, 2); /* Goto IDLE */
Id = __jtagRawRead(BitPos, 32); /* Read the Id */
__message "JTAG Id: ", Id:%x, "\n";
}
```

__jtagRawSync

Syntax `__jtagRawSync()`

Return value `int 0`

For use with The C-SPY J-Link/J-Trace driver

Description Sends arbitrary data to the JTAG interface. All accumulated bits using `__jtagRawWrite` will be written to the JTAG scan chain. The data is sent synchronously with TCK and typically sampled by the device on rising edge of TCK.

Example

The following piece of pseudocode illustrates how the data is written to the JTAG (on the TMS and TDI pins) and read (from TDO):

```
int i;
U32 tdo;
for (i = 0; i < numBits; i++) {
    TDI = tdi & 1; /* Set TDI pin */
    TMS = tms & 1; /* Set TMS pin */
    TCK = 0;
    TCK = 1;
    tdo <<= 1;
    if (TDO) {
        tdo |= 1;
    }
    tdi >>= 1;
    tms >>= 1;
}
```

__jtagRawWrite**Syntax**

```
__jtagRawWrite(tdi, tms, numbits)
```

Parameters

<i>tdi</i>	The data output to the TDI pin. This data is sent with the least significant bit first.
<i>tms</i>	The data output to the TMS pin. This data is sent with the least significant bit first.
<i>numbits</i>	The number of bits to transfer. Every bit results in a falling and rising edge of the JTAG TCK line. The maximum value is 64.

Return value

Returns the bit position of the data in the accumulated packet. Typically, this value is used when reading data from the JTAG.

For use with

The C-SPY J-Link/J-Trace driver

Description

Accumulates bits to be transferred to the JTAG. If 32 bits are not enough, this function can be called multiple times. Both data output lines (TMS and TDI) can be controlled separately.

Example

```
/* Send five 1 bits on TMS to go to TAP-RESET state */
__jtagRawWrite(0x1F, 0, 5); /* Store bits in buffer */
__jtagRawSync(); /* Transfer buffer, writing tms, tdi,
                  reading tdo */
```

Returns the JTAG ID of the ARM target device.

__jtagResetTRST

Syntax

```
__jtagResetTRST()
```

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 32: *__jtagResetTRST* return values

For use with

The C-SPY J-Link/J-Trace driver

Description

Resets the ARM TAP controller via the TRST JTAG signal.

__loadImage

Syntax

```
__loadImage(path, offset, debugInfoOnly)
```

Parameters

path

A string that identifies the path to the image to download. The path must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide for ARM*.

offset

An integer that identifies the offset to the destination address for the downloaded image.

debugInfoOnly

A non-zero integer value if no code or data should be downloaded to the target system, which means that C-SPY will only read the debug information from the debug file. Or, 0 (zero) for download.

Return value

Value	Result
Non-zero integer number	A unique module identification.
int 0	Loading failed.

Table 33: `__loadImage` return values

For use with

All C-SPY drivers.

Description

Loads an 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 501 and *Loading multiple images*, page 53.**`__memoryRestore`**

Syntax

```
__memoryRestore(zone, filename)
```

Parameters

*zone*A string that specifies the memory zone, see *C-SPY memory zones*, page 163.

	<i>filename</i>
	A string that specifies the file to be read. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide for ARM</i> .
Return value	int 0
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_memory.hex");</code>
See also	<i>Memory Restore dialog box</i> , page 172.

__memorySave

Syntax	<code>__memorySave(start, stop, format, filename)</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 ARM</i>.</p>

Return value	<code>int 0</code>
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");</pre>
See also	<i>Memory Save dialog box</i> , page 171.

__messageBoxYesCancel

Syntax `__messageBoxYesCancel(string message, string caption)`

Parameters

message
A message that will appear in the message box.

caption
The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 34: *__messageBoxYesCancel* return values

For use with All C-SPY drivers.

Description Displays a Yes/Cancel dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

__messageBoxYesNo

Syntax `__messageBoxYesNo(string message, string caption)`

Parameters

message
A message that will appear in the message box.

caption
The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 35: `__messageBoxYesNo` return values

For use with

All C-SPY drivers.

Description

Displays a Yes/No dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

`__openFile`

Syntax

```
__openFile(filename, access)
```

Parameters

filename

The file to be opened. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide for ARM*.

access

The access type (string).

These are mandatory but mutually exclusive:

"a" append, new data will be appended at the end of the open file

"r" read (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

Table 36: `__openFile` return values

Result	Value
Unsuccessful	An invalid file handle, which tests as False

Table 36: `__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 <code>__openFile</code> can specify a location relative to this directory. In addition, you can use argument variables such as <code>\$PROJ_DIR\$</code> and <code>\$TOOLKIT_DIR\$</code> in the path argument.
Example	<pre> __var myFileHandle; /* The macro variable to contain */ /* the file handle */ myFileHandle = __openFile("\$PROJ_DIR\$\Debug\Exe\test.tst", "r"); if (myFileHandle) { /* successful opening */ } </pre>
See also	For information about argument variables, see the <i>IDE Project Management and Building Guide for ARM</i> .

`__orderInterrupt`

Syntax	<pre> __orderInterrupt(<i>specification</i>, <i>first_activation</i>, <i>repeat_interval</i>, <i>variance</i>, <i>infinite_hold_time</i>, <i>hold_time</i>, <i>probability</i>) </pre>
Parameters	<p><i>specification</i></p> <p>The interrupt (string). The specification can either be the full specification used in the device description file (<code>ddf</code>) or only the name. In the latter case the interrupt system will automatically get the description from the device description file.</p> <p><i>first_activation</i></p> <p>The first activation time in cycles (integer)</p> <p><i>repeat_interval</i></p> <p>The periodicity in cycles (integer)</p> <p><i>variance</i></p> <p>The timing variation range in percent (integer between 0 and 100)</p>

	<i>infinite_hold_time</i>
	1 if infinite, otherwise 0.
	<i>hold_time</i>
	The hold time (integer)
	<i>probability</i>
	The probability in percent (integer between 0 and 100)
Return value	The macro returns an interrupt identifier (unsigned long). If the syntax of <i>specification</i> is incorrect, it returns -1.
For use with	The C-SPY Simulator.
Description	Generates an interrupt.
Example	This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles: <pre>__orderInterrupt("IRQ", 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 349.

__readAPReg

Syntax	<code>__readAPReg(<i>register</i>)</code>
--------	---

Parameters *register* An 8-bit AP register offset.

Return value

Result	Value
Successful	true
Unsuccessful	false

Table 37: *__readAPReg* return values

For use with
 The C-SPY I-jet/JTAGjet driver
 The C-SPY J-Link/J-Trace driver
 The C-SPY TI Stellaris driver

Description Performs a read operation from an AP register of the currently selected access port.

__readDPRReg

Syntax *__readDPRReg(register)*

Parameters *register* An 8-bit DP register offset.

Return value

Result	Value
Successful	true
Unsuccessful	false

Table 38: *__readDPRReg* return values

For use with
 The C-SPY I-jet/JTAGjet driver
 The C-SPY J-Link/J-Trace driver
 The C-SPY TI Stellaris driver

Description Performs a read operation from a DP register.

__readFile

Syntax `__readFile(fileHandle, valuePtr)`

Parameters

fileHandle

A macro variable used as filehandle by the `__openFile` macro.

valuePtr

A pointer to a variable.

Return value

Result	Value
Successful	0
Unsuccessful	Non-zero error number

Table 39: `__readFile` return values

For use with

All C-SPY drivers.

Description

Reads a sequence of hexadecimal digits from the given file and converts them to an unsigned long which is assigned to the *value* parameter, which should be a pointer to a macro variable.

Only printable characters representing hexadecimal digits and white-space characters are accepted, no other characters are allowed.

Example

```
__var number;
if (__readFile(myFileHandle, &number) == 0)
{
    // Do something with number
}
```

In this example, if the file pointed to by `myFileHandle` contains the ASCII characters `1234 abcd 90ef`, consecutive reads will assign the values `0x1234 0xabcd 0x90ef` to the variable `number`.

__readFileByte

Syntax

`__readFileByte(fileHandle)`

Parameters

fileHandle

A macro variable used as filehandle by the `__openFile` macro.

Return value

-1 upon error or end-of-file, otherwise a value between 0 and 255.

For use with	All C-SPY drivers.
Description	Reads one byte from a file.
Example	<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 163.</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	<pre> __readMemory8(0x0108, " "); </pre>

__readMemory16

Syntax	<pre> __readMemory16(<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 163.</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 `__readMemory16(0x0108, "");`

__readMemory32

Syntax `__readMemory32(address, zone)`

Parameters

address

The memory address (integer).

zone

A string that specifies the memory zone, see *C-SPY memory zones*, page 163.

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 `__readMemory32(0x0108, "");`

__registerMacroFile

Syntax `__registerMacroFile(filename)`

Parameters

filename

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 *IDE Project Management and Building Guide for ARM*.

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 `__registerMacroFile("c:\\testdir\\macro.mac");`

See also *Using C-SPY macros*, page 367.

__resetFile

Syntax	<code>__resetFile(<i>fileHandle</i>)</code>
Parameters	<i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Rewinds a file previously opened by <code>__openFile</code> .

__restoreSoftwareBreakpoints

Syntax	<code>__restoreSoftwareBreakpoints()</code>
Return value	<code>int 0</code>
For use with	The C-SPY I-jet/JTAGjet driver The C-SPY J-Link/J-Trace driver The C-SPY TI Stellaris driver The C-SPY Macraigor driver
Description	Restores automatically any breakpoints that were destroyed during system startup. This can be useful if you have an application that is copied to RAM during startup and is then executing in RAM. This can, for example, be the case if you use the <code>initialize</code> by <code>copy</code> directive for code in the linker configuration file or if you have any <code>__ramfunc</code> declared functions in your application. In this case, any breakpoints will be overwritten during the RAM copying when the application execution starts. By using the this macro, C-SPY will restore the destroyed breakpoints.

__selectCore

Syntax	<code>__selectCore(int core)</code>
Parameters	<p><i>core</i></p> <p>The core to switch to. The cores are numbered from 0 and upwards.</p>
Return value	<code>int 0</code>
For use with	<p>The C-SPY simulator.</p> <p>The C-SPY I-jet/JTAGjet driver.</p>
Description	Switches focus from the current core to the specified core for the duration of the macro invocation or until any next invocation of <code>__selectCore</code> .
Example	<pre>test () { __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n"; __selectCore(0); __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n"; __selectCore(1); __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n"; }</pre> <p>A typical result of the above macro would be (assuming that the original core was number 1):</p> <pre>Core: 1 pc = 0000213C Core: 0 pc = 00000494 Core: 1 pc = 0000213C</pre>
See also	<code>__getSelectedCore</code> , page 392.

__setCodeBreak

Syntax	<code>__setCodeBreak(location, count, condition, cond_type, action)</code>
Parameters	<p><i>location</i></p> <p>A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see <i>Enter Location dialog box</i>, page 157.</p>

count

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).

condition

The breakpoint condition (string).

cond_type

The condition type; either "CHANGED" or "TRUE" (string).

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 40: `__setCodeBreak` return values

For use with

The C-SPY hardware debugger 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", "");
```

See also

Breakpoints, page 125.

__setDataBreak

Syntax

In the simulator:

```
__setDataBreak(location, count, condition, cond_type, access,
action)
```

In the I-jet/JTAGjet driver:

```
__setDataBreak(location, access, extend, match, data, mask)
```


Parameters

location

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see *Enter Location dialog box*, page 157.

count

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).

This parameter applies to the simulator only.

condition

The breakpoint condition (string).

This parameter applies to the simulator only.

cond_type

The condition type; either "CHANGED" or "TRUE" (string).

This parameter applies to the simulator only.

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.

This parameter applies to the simulator only.

extend

Extends the breakpoint so that a whole data structure is covered. For data structures that do not fit the size of the possible breakpoint ranges supplied by the hardware breakpoint unit, for example three bytes, the breakpoint range will not cover the whole data structure. Note that the breakpoint range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data. Choose between "TRUE" or "FALSE".

This parameter applies to I-jet/JTAGjet only.

match

Enables matching of the accessed data. Choose between "TRUE" or "FALSE".

This parameter applies to I-jet/JTAGjet only.

data

A data value to match, in unsigned 32-bit format.

This parameter applies to I-jet/JTAGjet only.

mask

Specifies which part of the data value to match (word, halfword, or byte), in unsigned 32-bit format.

This parameter applies to I-jet/JTAGjet only.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 41: `__setDataBreak` return values

For use with

The I-jet/JTAGjet driver

Description

Sets a data breakpoint, that is, a breakpoint which is triggered directly after the processor has read or written data at the specified location.

Example

For the C-SPY simulator:

```
__var brk;
brk = __setDataBreak(":0x4710", 3, "d>6", "TRUE",
    "W", "ActionData()");
...
__clearBreak(brk);
```

See also

Breakpoints, page 125.

`__setDataLogBreak`

Syntax

```
__setDataLogBreak(variable, access)
```

Parameters

variable

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.

access

The memory access type: "R", for read, "W" for write, or "RW" for read/write.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 42: `__setDataLogBreak` return values

For use with

The C-SPY Simulator.
The C-SPY I-jet/JTAGjet driver.

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 125 and *Getting started using data logging*, page 97.

`__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 157.

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 43: `__setLogBreak` return values

For use with

The C-SPY hardware debugger 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 375 and *Breakpoints*, page 125.

__setSimBreak

Syntax

```
__setSimBreak(location, access, action)
```

Parameters

location

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see *Enter Location dialog box*, page 157.

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 44: `__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

In the simulator:

```
__setTraceStartBreak(location)
```

In the I-jet/JTAGjet driver:

```
__setTraceStartBreak(location, access, extend, match, data, mask)
```

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 157.

access

The memory access type: "F" for fetch, "R" for read, "W" for write, or "RW" for read/write.

This parameter applies to I-jet/JTAGjet only.

extend

Extends the breakpoint so that a whole data structure is covered. For data structures that do not fit the size of the possible breakpoint ranges supplied by the hardware breakpoint unit, for example three bytes, the breakpoint range will not cover the whole data structure. Note that the breakpoint range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data. Choose between "TRUE" or "FALSE".

This parameter applies to I-jet/JTAGjet only.

match

Enables matching of the accessed data. Choose between "TRUE" or "FALSE".

This parameter applies to I-jet/JTAGjet only.

data

A data value to match, in unsigned 32-bit format.

This parameter applies to I-jet/JTAGjet only.

mask

Specifies which part of the data value to match (word, halfword, or byte), in unsigned 32-bit format.

This parameter applies to I-jet/JTAGjet only.

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 45: `__setTraceStartBreak` return values

For use with

The C-SPY Simulator.

	The C-SPY I-jet/JTAGjet driver.
Description	Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is started.
Example	<pre> __var startTraceBp; __var stopTraceBp; traceOn() { startTraceBp = __setTraceStartBreak ("C:\\TEMP\\Utilities.c).23.1"); stopTraceBp = __setTraceStopBreak ("C:\\temp\\Utilities.c).30.1"); } traceOff() { __clearBreak(startTraceBp); __clearBreak(stopTraceBp); } </pre>
See also	<i>Breakpoints</i> , page 125.

__setTraceStopBreak

Syntax	<p>In the simulator:</p> <pre>__setTraceStopBreak(<i>location</i>)</pre> <p>In the I-jet/JTAGjet driver:</p> <pre>__setTraceStopBreak(<i>location</i>, <i>access</i>, <i>extend</i>, <i>match</i>, <i>data</i>, <i>mask</i>)</pre>
Parameters	<p><i>location</i></p> <p>A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see <i>Enter Location dialog box</i>, page 157.</p> <p><i>access</i></p> <p>The memory access type: "F" for fetch, "R" for read, "W" for write, or "RW" for read/write.</p> <p>This parameter applies to I-jet/JTAGjet only.</p>

extend

Extends the breakpoint so that a whole data structure is covered. For data structures that do not fit the size of the possible breakpoint ranges supplied by the hardware breakpoint unit, for example three bytes, the breakpoint range will not cover the whole data structure. Note that the breakpoint range will be extended beyond the size of the data structure, which might cause false triggers at adjacent data. Choose between "TRUE" or "FALSE".

This parameter applies to I-jet/JTAGjet only.

match

Enables matching of the accessed data. Choose between "TRUE" or "FALSE".

This parameter applies to I-jet/JTAGjet only.

data

A data value to match, in unsigned 32-bit format.

This parameter applies to I-jet/JTAGjet only.

mask

Specifies which part of the data value to match (word, halfword, or byte), in unsigned 32-bit format.

This parameter applies to I-jet/JTAGjet only.

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 46: *__setTraceStopBreak* return values

For use with

The C-SPY Simulator.
The C-SPY I-jet/JTAGjet driver.

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is stopped.

Example

See *__setTraceStartBreak*, page 421.

See also

Breakpoints, page 125.

__sourcePosition

Syntax	<code>__sourcePosition(<i>linePtr</i>, <i>colPtr</i>)</code>
Parameters	<p><i>linePtr</i> Pointer to the variable storing the line number</p> <p><i>colPtr</i> Pointer to the variable storing the column number</p>

Return value

Result	Value
Successful	Filename string
Unsuccessful	Empty (" ") string

Table 47: __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	<code>__strFind(<i>macroString</i>, <i>pattern</i>, <i>position</i>)</code>
Parameters	<p><i>macroString</i> A macro string.</p> <p><i>pattern</i> The string pattern to search for</p> <p><i>position</i> The position where to start the search. The first position is 0</p>
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 `__strFind("Compiler", "pile", 0) = 3`
 `__strFind("Compiler", "foo", 0) = -1`

See also *Macro strings*, page 374.

__subString

Syntax `__subString(macroString, position, length)`

Parameters *macroString*

A macro string.

position

The start position of the substring. The first position is 0.

length

The length of the substring

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 (*macroString*).

Example `__subString("Compiler", 0, 2)`

The resulting macro string contains Co.

`__subString("Compiler", 3, 4)`

The resulting macro string contains pile.

See also *Macro strings*, page 374.

__targetDebuggerVersion

Syntax `__targetDebuggerVersion()`

Return value A string that represents the version number of the C-SPY debugger processor module.

For use with All C-SPY drivers.

Description This macro returns the version number of the C-SPY debugger processor module.

Example

```
__var toolVer;
toolVer = __targetDebuggerVersion();
__message "The target debugger version is, ", toolVer;
```

__toLower

Syntax `__toLower(macroString)`

Parameters *macroString*
A macro string.

Return value The converted macro string.

For use with All C-SPY drivers.

Description This macro returns a copy of the parameter *macroString* where all the characters have been converted to lower case.

Example

```
__toLower("IAR")
```

The resulting macro string contains `iar`.

```
__toLower("Mix42")
```

The resulting macro string contains `mix42`.

See also *Macro strings*, page 374.

__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 * hptr = "Hello World!";
```

this macro call:

```
__toString(hptr, 5)
```

would return the macro string containing Hello.

See also *Macro strings*, page 374.

__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 *macroString* where all the characters have been converted to upper case.

Example `__toUpper("string")`
The resulting macro string contains `STRING`.

See also *Macro strings*, page 374.

__unloadImage

Syntax `__unloadImage(module_id)`

Parameters *module_id*
An integer which represents a unique module identification, which is retrieved as a return value from the corresponding `__loadImage` C-SPY macro.

Return value

Value	Result
<i>module_id</i>	A unique module identification (the same as the input parameter).
int 0	The unloading failed.

Table 48: *__unloadImage* return values

For use with

All C-SPY drivers.

Description

Unloads debug information from an already downloaded image.

See also

Loading multiple images, page 53 and *Images*, page 501.**__writeAPReg**

Syntax

`__writeAPReg(data, register)`

Parameters

<i>data</i>	A 32-bit value.
<i>register</i>	An 8-bit AP register offset.

Return value

Result	Value
Successful	true
Unsuccessful	false

Table 49: *__writeAPReg* return values

For use with

The C-SPY I-Jet/JTAGjet driver
 The C-SPY J-Link/J-Trace driver
 The C-SPY TI Stellaris driver

Description

Performs a write operation to an AP register of the currently selected access port.

__writeDPRReg

Syntax

`__writeDPRReg(data, register)`

Parameters

<i>data</i>	A 32-bit value.
-------------	-----------------

register An 8-bit DP register offset.

Return value

Result	Value
Successful	true
Unsuccessful	false

Table 50: `__writeDPReg` return values

For use with

The C-SPY I-Jet/JTAGjet driver
 The C-SPY J-Link/J-Trace driver
 The C-SPY TI Stellaris driver

Description

Performs a write operation to a DP register.

Example

```
__writeDPReg(0x010000F0, 0x8)
/* Selects access port 1 and bank 15 */
```

__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	<i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro. <i>value</i> 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 163.
Return value	int 0
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 163.
Return value	int 0
For use with	All C-SPY drivers.
Description	Writes two bytes to a given memory location.
Example	<code>__writeMemory16(0x2FFF, 0x8020, "");</code>

__writeMemory32

Syntax	<code>__writeMemory32(value, address, zone)</code>
Parameters	<p><i>value</i></p> <p>An integer.</p> <p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 163.</p>
Return value	int 0
For use with	All C-SPY drivers.
Description	Writes four bytes to a given memory location.
Example	<code>__writeMemory32(0x5555FFFF, 0x8020, "");</code>

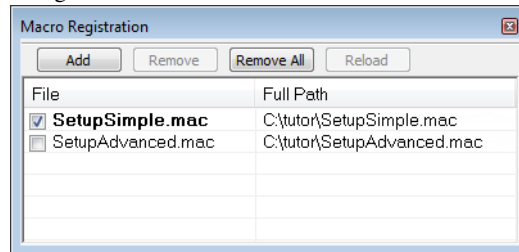
Graphical environment for macros

Reference information about:

- *Macro Registration window*, page 433
- *Debugger Macros window*, page 435
- *Macro Quicklaunch window*, page 437

Macro Registration window

The Macro Registration window is available from the **View>Macros** submenu during a debug session.



Use this window to list, register, and edit your debugger macro files.

Double-click a macro file to open it in the editor window and edit it.

Requirements

None; this window is always available.

Display area

This area contains these columns:

File

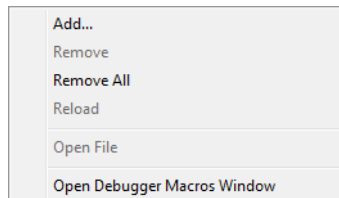
The name of an available macro file. To register the macro file, select the check box to the left of the filename. The name of a registered macro file appears in bold style.

Full path

The path to the location of the added macro file.

Context menu

This context menu is available:



These commands are available:

Add

Opens a file browser where you can locate the macro file that you want to add to the list. This menu command is also available as a function button at the top of the window.

Remove

Removes the selected debugger macro file from the list. This menu command is also available as a function button at the top of the window.

Remove All

Removes all macro files from the list. This menu command is also available as a function button at the top of the window.

Reload

Registers the selected macro file. Typically, this is useful when you have edited a macro file. This menu command is also available as a function button at the top of the window.

Open File

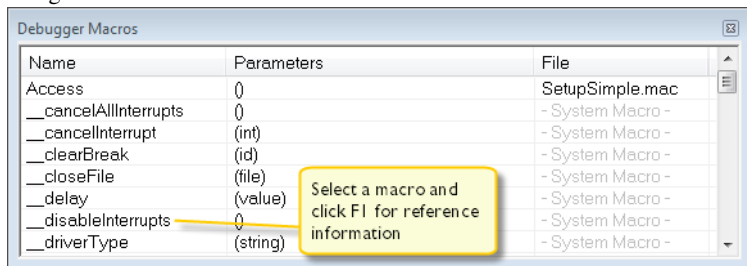
Opens the selected macro file in the editor window.

Open Debugger Macros Window

Opens the Debugger Macros window.

Debugger Macros window

The Debugger Macros window is available from the **View>Macro** submenu during a debug session.



Use this window to list all registered debugger macro functions, either predefined system macros or your own. This window is useful when you edit your own macro functions and want an overview of all available macros that you can use.

Double-clicking a macro defined in a file opens that file in the editor window.

To open a macro in the **Macro Quicklaunch** window, drag it from the **Debugger Macros** window and drop it in the **Macro Quicklaunch** window.

Select a macro and press F1 to get online help information for that macro.

Requirements

None; this window is always available.

Display area

This area contains these columns:

Name

The name of the debugger macro.

Parameters

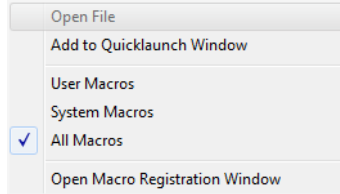
The parameters of the debugger macro.

File

For macros defined in a file, the name of the file is displayed. For predefined system macros, -System Macro- is displayed.

Context menu

This context menu is available:



These commands are available:

Open File

Opens the selected debugger macro file in the editor window.

Add to Quicklaunch Window

Adds the selected macro to the **Macro Quicklaunch** window.

User Macros

Lists only the debugger macros that you have defined yourself.

System Macros

Lists only the predefined system macros.

All Macros

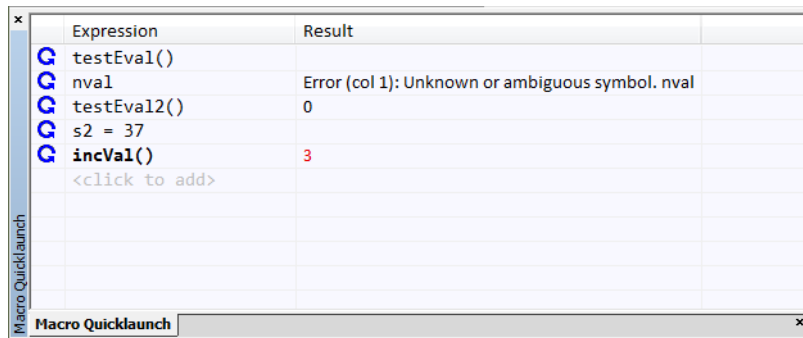
Lists all debugger macros, both predefined system macros and your own.

Open Macro Registration Window

Opens the **Macro Registration** window.

Macro Quicklaunch window

The **Macro Quicklaunch** window is available from the **View** menu.



Expression	Result
testEval()	
nval	Error (col 1): Unknown or ambiguous symbol. nval
testEval2()	0
s2 = 37	
incVal()	3
<click to add>	

Use this window to evaluate expressions, typically C-SPY macros.

For some devices, there are predefined C-SPY macros available with device support, typically provided by the chip manufacturer. These macros are useful for performing certain device-specific tasks. The macros are available in the **Macro Quicklaunch** window and are easily identified by their green icon,


The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is primarily designed for evaluating C-SPY macros. The window gives you precise control over when to evaluate an expression.

To add an expression:

- I Choose one of these alternatives:
 - Drag the expression to the window
 - In the **Expression** column, type the expression you want to examine.

If the expression you add and want to evaluate is a C-SPY macro, the macro must first be registered, see *Using C-SPY macros*, page 367.

To evaluate an expression:

- I  Double-click the **Recalculate** icon to calculate the value of that expression.

Requirements

None; this window is always available.

Display area

This area contains these columns:



Recalculate icon

To evaluate the expression, double-click the icon. The latest evaluated expression appears in bold style.

Expression

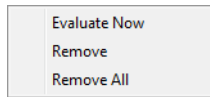
One or several expressions that you want to evaluate. Click <click to add> to add an expression. If the return value has changed since last time, the value will be displayed in red.

Result

Shows the return value from the expression evaluation.

Context menu

This context menu is available:



These commands are available:

Evaluate Now

Evaluates the selected expression.

Remove

Removes the selected expression.

Remove All

Removes all selected expressions.

The C-SPY command line utility—`cspybat`

- Using C-SPY in batch mode
- Summary of C-SPY command line options
- Reference information on C-SPY command line options.

Using C-SPY in batch mode

You can execute C-SPY in batch mode if you use the command line utility `cspybat`, installed in the directory `common\bin`.

These topics are covered:

- Starting `cspybat`
- Output
- Invocation syntax

STARTING CSPYBAT

- 1 To start `cspybat` you must first create a batch file. An easy way to do that is to use one of the batch files that C-SPY automatically generates when you start C-SPY in the IDE.

C-SPY generates a batch file `projectname.buildconfiguration.cspy.bat` every time C-SPY is initialized. In addition, two more files are generated:

- `project.buildconfiguration.general.xml`, which contains options specific to `cspybat`.
- `project.buildconfiguration.driver.xml`, which contains options specific to the C-SPY driver you are using.

You can find the files in the directory `$PROJ_DIR$\settings`. The files contain the same settings as the IDE, and provide hints about additional options that you can use.

- 2 To start `cspybat`, you can use this command line:

```
project.cspybat.bat [debugfile]
```

Note that *debugfile* is optional. You can specify it if you want to use a different debug file than the one that is used in the *project.buildconfiguration.general.xml* file.

OUTPUT

When you run `cspybat`, these types of output can be produced:

- Terminal output from `cspybat` itself

All such terminal output is directed to `stderr`. Note that if you run `cspybat` from the command line without any arguments, the `cspybat` version number and all available options including brief descriptions are directed to `stdout` and displayed on your screen.
- Terminal output from the application you are debugging

All such terminal output is directed to `stdout`, provided that you have used the `--plugin` option. See *--plugin*, page 481.
- Error return codes

`cspybat` returns status information to the host operating system that can be tested in a batch file. For *successful*, the value `int 0` is returned, and for *unsuccessful* the value `int 1` is returned.

INVOCATION SYNTAX

The invocation syntax for `cspybat` is:

```
cspybat processor_DLL driver_DLL debug_file
        [cspybat_options] --backend driver_options
```

Note: In those cases where a filename is required—including the DLL files—you are recommended to give a full path to the filename.

Parameters

The parameters are:

Parameter	Description
<i>processor_DLL</i>	The processor-specific DLL file; available in <code>arm\bin</code> .
<i>driver_DLL</i>	The C-SPY driver DLL file; available in <code>arm\bin</code> .
<i>debug_file</i>	The object file that you want to debug (filename extension <code>out</code>). See also <i>--debugfile</i> , page 450.
<i>cspybat_options</i>	The command line options that you want to pass to <code>cspybat</code> . Note that these options are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 449.

Table 51: *cspybat* parameters

Parameter	Description
<code>--backend</code>	Marks the beginning of the parameters to the C-SPY driver; all options that follow will be sent to the driver. Note that this option is mandatory.
<code>driver_options</code>	The command line options that you want to pass to the C-SPY driver. Note that some of these options are mandatory and some are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 449.

Table 51: cspybat parameters (Continued)

Summary of C-SPY command line options

Reference information about:

- General cspybat options
- Options available for all C-SPY drivers
- Options available for the simulator driver
- Options available for the C-SPY Angel debug monitor driver
- Options available for the C-SPY GDB Server driver
- Options available for the C-SPY IAR ROM-monitor driver
- Options available for the C-SPY I-jet/JTAGjet driver
- Options available for the C-SPY CMSIS-DAP driver
- Options available for the C-SPY J-Link/J-Trace driver
- Options available for the C-SPY TI Stellaris driver
- Options available for the C-SPY TI XDS driver
- Options available for the C-SPY Macraigor driver
- Options available for the C-SPY RDI driver
- Options available for the C-SPY ST-LINK driver
- Options available for the C-SPY third-party drivers

GENERAL CSPYBAT OPTIONS

<code>--backend</code>	Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).
<code>--code_coverage_file</code>	Enables the generation of code coverage information and places it in a specified file.
<code>--cycles</code>	Specifies the maximum number of cycles to run.

<code>--debugfile</code>	Specifies an alternative debug file.
<code>--download_only</code>	Downloads a code image without starting a debug session afterwards.
<code>-f</code>	Extends the command line.
<code>--flash_loader</code>	Specifies a flash loader specification XML file.
<code>--leave_running</code>	Starts the execution on the target and then exits but leaves the target running.
<code>--macro</code>	Specifies a macro file to be used.
<code>--macro_param</code>	Assigns a value to a C-SPY macro parameter.
<code>--plugin</code>	Specifies a plugin file to be used.
<code>--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>--BE8</code>	Uses the big-endian format BE8. For reference information, see the <i>IAR C/C++ Development Guide for ARM</i> .
<code>--BE32</code>	Uses the big-endian format BE32. For reference information, see the <i>IAR C/C++ Development Guide for ARM</i> .
<code>--cpu</code>	Specifies a processor variant. For reference information, see the <i>IAR C/C++ Development Guide for ARM</i> .
<code>--device</code>	Specifies the name of the device.

`--drv_attach_to_program` Attaches the debugger to a running application at its current location. For reference information, see *Download*, page 500, specifically the option **Attach to running target**.

`--drv_catch_exceptions` Makes the application stop for certain exceptions.

`--drv_communication` Specifies the communication link to be used.

`--drv_communication_log` Creates a log file.

`--drv_default_breakpoint` Sets the type of breakpoint resource to be used when setting breakpoints.

`--drv_reset_to_cpu_start` Omits setting the PC when resetting the application.

`--drv_restore_breakpoints` Restores automatically any breakpoints that were destroyed during system startup.

`--drv_suppress_download` Suppresses download of the executable image. For reference information, see *Download*, page 500, specifically the option **Suppress download**.

`--drv_vector_table_base` Specifies the location of the Cortex-M reset vector and the initial stack pointer value.

`--drv_verify_download` Verifies the target program. For reference information, see *Download*, page 500, specifically the option **Verify download**.

Available for Angel, GDB Server, IAR ROM-monitor, J-Link/J-Trace, JTAGjet, Macraigor, RDI, ST-LINK, TI Stellaris, and TI XDS.

`--endian` Specifies the byte order of the generated code and data. For reference information, see the *IAR C/C++ Development Guide for ARM*.

`--fpu` Selects the type of floating-point unit. For reference information, see the *IAR C/C++ Development Guide for ARM*.

`--leave_running` Starts the execution on the target and then exits but leaves the target running.

`-p` Specifies the device description file to be used.

`--proc_stack_stack` Provides C-SPY with information about reserved stacks.
`--semihosting` Enables semihosted I/O.

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

`--disable_interrupts` Disables the interrupt simulation.
`--mapu` Activates memory access checking.

OPTIONS AVAILABLE FOR THE C-SPY ANGEL DEBUG MONITOR DRIVER

`--drv_catch_exceptions` Makes the application stop for certain exceptions.
`--rdi_heartbeat` Makes C-SPY poll your target system periodically. For reference information, see *Angel*, page 506, specifically the option **Send heartbeat**.
`--rdi_step_max_one` Executes one instruction.

OPTIONS AVAILABLE FOR THE C-SPY GDB SERVER DRIVER

`--drv_default_breakpoint` Sets the type of breakpoint resource to be used when setting breakpoints.
`--gdbserv_exec_command` Sends a command string to the GDB Server.

OPTIONS AVAILABLE FOR THE C-SPY IAR ROM-MONITOR DRIVER

There are no additional options specific to the C-SPY IAR ROM-monitor driver.

OPTIONS AVAILABLE FOR THE C-SPY I-JET/JTAGJET DRIVER

`--drv_catch_exceptions` Makes the application stop for certain exceptions.
`--drv_default_breakpoint` Sets the type of breakpoint resource to be used when setting breakpoints.

<code>--drv_interface</code>	Selects the communication interface.
<code>--drv_interface_speed</code>	Specifies the JTAG and SWD speed.
<code>--jet_board_cfg</code>	Specifies a probe configuration file.
<code>--jet_board_did</code>	Selects which CPU to debug on a multi-core system.
<code>--jet_cpu_clock</code>	Specifies the frequency of the internal processor clock.
<code>--jet_ir_length</code>	Specifies the number of IR bits preceding the ARM core to connect to.
<code>--jet_power_from_probe</code>	Specifies the power supply from the I-jet or I-jet Trace probe.
<code>--jet_probe</code>	Specifies which debug system the C-SPY I-jet/JTAGjet driver is an interface to.
<code>--jet_script_file</code>	Specifies the reset script file.
<code>--jet_standard_reset</code>	Selects the reset strategy to be used when C-SPY starts.
<code>--jet_startup_connection_timeout</code>	Prolongs the time that the C-SPY driver tries to connect to the target board.
<code>--jet_swo_on_d0</code>	Specifies that SWO trace data is output on the trace data pin D0
<code>--jet_swo_prescaler</code>	Specifies the SWO prescaler for the CPU clock frequency.
<code>--jet_swo_protocol</code>	Selects the SWO communication protocol.
<code>--jet_tap_position</code>	Selects a specific device in the JTAG scan chain.
<code>--reset_style</code>	Specifies the reset strategies that will be available when debugging.

OPTIONS AVAILABLE FOR THE C-SPY CMSIS-DAP DRIVER

<code>--drv_catch_exceptions</code>	Makes the application stop for certain exceptions.
-------------------------------------	--

<code>--drv_default_breakpoint</code>	Sets the type of breakpoint resource to be used when setting breakpoints.
<code>--drv_interface</code>	Selects the communication interface.
<code>--drv_interface_speed</code>	Specifies the JTAG and SWD speed.
<code>--jet_board_cfg</code>	Specifies a probe configuration file.
<code>--jet_board_did</code>	Selects which CPU to debug on a multi-core system.
<code>--jet_probe</code>	Specifies which debug system the C-SPY driver is an interface to.
<code>--jet_script_file</code>	Specifies the reset script file.
<code>--jet_standard_reset</code>	Selects the reset strategy to be used when C-SPY starts.
<code>--jet_startup_connection_timeout</code>	Prolongs the time that the C-SPY driver tries to connect to the target board.
<code>--jet_tap_position</code>	Selects a specific device in the JTAG scan chain.
<code>--reset_style</code>	Specifies the reset strategies that will be available when debugging.

OPTIONS AVAILABLE FOR THE C-SPY J-LINK/J-TRACE DRIVER

<code>--drv_catch_exceptions</code>	Makes the application stop for certain exceptions.
<code>--drv_default_breakpoint</code>	Sets the type of breakpoint resource to be used when setting breakpoints.
<code>--drv_interface</code>	Selects the communication interface.
<code>--drv_interface_speed</code>	Specifies the JTAG and SWD speed.
<code>--drv_swo_clock_setup</code>	Specifies the CPU clock and the wanted SWO speed.
<code>--jlink_dcc_timeout</code>	Specifies the timeout for a pending request from C-SPY to the DCC agent on target.
<code>--jlink_device_select</code>	Selects a specific device in the JTAG scan chain.

<code>--jlink_exec_command</code>	Calls the <code>__jlinkExecCommand</code> macro after target connection has been established.
<code>--jlink_initial_speed</code>	Sets the initial JTAG communication speed in kHz.
<code>--jlink_ir_length</code>	Sets the number of IR bits preceding the ARM core to connect to.
<code>--jlink_reset_strategy</code>	Selects the reset strategy to be used at debugger startup.
<code>--jlink_script_file</code>	Specifies the script file for setting up hardware.
<code>--jlink_trace_source</code>	Selects either ETB or ETM as the trace source.

OPTIONS AVAILABLE FOR THE C-SPY TI STELLARIS DRIVER

<code>--drv_interface</code>	Selects the communication interface.
<code>--drv_interface_speed</code>	Specifies the JTAG and SWD speed.

OPTIONS AVAILABLE FOR THE C-SPY TI XDS DRIVER

<code>--drv_catch_exceptions</code>	Makes the application stop for certain exceptions.
<code>--drv_default_breakpoint</code>	Sets the type of breakpoint resource to be used when setting breakpoints.
<code>--drv_interface</code>	Selects the communication interface.
<code>--drv_interface_speed</code>	Specifies the JTAG and SWD speed.
<code>--xds_board_file</code>	Overrides the default board file.
<code>--xds_rootdir</code>	Specifies the installation directory of the TI XDS driver package.
<code>--xds_reset_strategy</code>	Specifies the reset strategy to use.

OPTIONS AVAILABLE FOR THE C-SPY MACRAIGOR DRIVER

<code>--drv_default_breakpoint</code>	Sets the type of breakpoint resource to be used when setting breakpoints.
---------------------------------------	---

<code>--drv_interface</code>	Selects the communication interface.
<code>--drv_interface_speed</code>	Specifies the JTAG and SWD speed.
<code>--mac_handler_address</code>	Specifies the location of the debug handler used by Intel XScale devices.
<code>--mac_jtag_device</code>	Selects the device corresponding to the hardware interface.
<code>--mac_multiple_targets</code>	Specifies the device to connect to, if there are more than one device on the JTAG scan chain.
<code>--mac_reset_pulls_reset</code>	Makes C-SPY generate an initial hardware reset.
<code>--mac_set_temp_reg_buffer</code>	Provides the driver with a physical RAM address for accessing the coprocessor.
<code>--mac_xscale_ir7</code>	Specifies that the XScale ir7 architecture is used.

OPTIONS AVAILABLE FOR THE C-SPY RDI DRIVER

<code>--drv_catch_exceptions</code>	Makes the application stop for certain exceptions.
<code>--rdi_allow_hardware_reset</code>	Performs a hardware reset.
<code>--rdi_driver_dll</code>	Specifies the path to the driver DLL file.
<code>--rdi_step_max_one</code>	Executes one instruction.

OPTIONS AVAILABLE FOR THE C-SPY ST-LINK DRIVER

<code>--drv_catch_exceptions</code>	Makes the application stop for certain exceptions.
<code>--drv_interface</code>	Selects the communication interface.
<code>--drv_interface_speed</code>	Specifies the JTAG and SWD interface speed.
<code>--drv_swo_clock_setup</code>	Specifies the CPU clock and the wanted SWO speed.
<code>--stlink_reset_strategy</code>	Specifies the reset strategy to use.

OPTIONS AVAILABLE FOR THE C-SPY THIRD-PARTY DRIVERS

For information about any options specific to the third-party driver you are using, see its documentation.

Reference information on C-SPY command line options

This section gives detailed reference information about each `cspybat` option and each option available to the C-SPY drivers.

--backend

Syntax	<code>--backend {driver options}</code>
Parameters	<i>driver options</i> Any option available to the C-SPY driver you are using.
For use with	<code>cspybat</code> (mandatory).
Description	Use this option to send options to the C-SPY driver. All options that follow <code>--backend</code> will be passed to the C-SPY driver, and will not be processed by <code>cspybat</code> itself.



This option is not available in the IDE.

--code_coverage_file

Syntax	<code>--code_coverage_file file</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>file</i> The name of the destination file for the code coverage information.
For use with	<code>cspybat</code>
Description	Use this option to enable the generation of 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 `stderr`.

See also

Code coverage, page 269, *--cycles*, page 450, *--timeout*, page 486.



To set this option, choose **View>Code Coverage**, right-click and choose **Save As** when the C-SPY debugger is running.

--cycles

Syntax

`--cycles cycles`

Note that this option must be placed before the `--backend` option on the command line.

Parameters

cycles

The number of cycles to run.

For use with

`cspybat`

Description

Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted. Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.



This option is not available in the IDE.

--debugfile

Syntax

`--debugfile filename`

Parameters

filename

The name of the debug file to use.

For use with

`cspybat`

This option can be placed both before and after the `--backend` option on the command line.


Description

Use this option to make `cspybat` use the specified debugfile instead of the one used in the generated `cpsybat.bat` file.




This option is not available in the IDE.

--device

Syntax	<code>--device=device_name</code>	
Parameters	<i>device_name</i>	The name of the device, for example, ADuC7030, AT91SAM7S256, LPC2378, STR912FM44, or TMS470R1B1M.
For use with	All C-SPY drivers.	
Description	Use this option to specify the name of the device.	
		To set related option, choose: Project>Options>General Options>Target>Device

--disable_interrupts

Syntax	<code>--disable_interrupts</code>	
For use with	The C-SPY Simulator driver.	
Description	Use this option to disable the interrupt simulation.	
		To set this option, choose Simulator>Interrupt Setup and deselect the Enable interrupt simulation option.

--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.	



Project>Download>Download active application

Alternatively, to set a related option, choose:

Project>Options>Debugger>Setup and deselect **Run to**.

--drv_catch_exceptions

Syntax

--drv_catch_exceptions=*value*

Parameters

value

(for ARM9, Cortex-R4,
ARM11, and Cortex-A)

A value in the range of 0–0x1FF. Each bit specifies which exception to catch:

Bit 0 = Reset

Bit 1 = Undefined instruction

Bit 2 = SWI

Bit 3 = Prefetch abort

Bit 4 = Data abort

Bit 5 = Not used

Bit 6 = IRQ

Bit 7 = FIQ

Bit 8 = Other errors

value

(for Cortex-M)

A value in the range of 0–0x7FF. Each bit specifies which exception to catch:

Bit 0 = CORERESSET - Reset Vector

Bit 4 = MMERR - Memory Management Fault

Bit 5 = NOCPERR - Coprocessor Access Error

Bit 6 = CHKERR - Checking Error

Bit 7 = STATERR - State Error


Bit 8 = BUSERR - Bus Error

Bit 9 = INTERR - Interrupt Service Errors

Bit 10 = HARDERR - Hard Fault

For use with

The C-SPY Angel debug monitor driver.

	The C-SPY I-jet/JTAGjet driver
	The C-SPY J-Link/J-Trace driver
	The C-SPY CMSIS-DAP driver
	The C-SPY RDI driver
	The C-SPY ST-LINK driver
	The C-SPY TI XDS driver.
Description	Use this option to make the application stop when a certain exception occurs.
See also	<i>Setting a breakpoint on an exception vector</i> , page 134.
	 For the C-SPY Angel debug monitor driver, use: Project>Options>Debugger>Extra Options
	For the C-SPY I-jet/JTAGjet driver and the C-SPY J-Link/J-Trace driver, use: Project>Options>Debugger>Driver>Breakpoints>Catch exceptions
	For the C-SPY RDI driver, use: Project>Options>Debugger>RDI>Catch exceptions

--drv_communication

Syntax	<code>--drv_communication=connection</code>
Parameters	Where connection is one of these for the C-SPY Angel debug monitor driver:
	Via Ethernet
	UDP: <i>ip_address</i>
	UDP: <i>ip_address, port</i>
	UDP: <i>hostname</i>
	UDP: <i>hostname, port</i>

Via serial port *port:baud,parity,stop_bit,handshake*

port = COM1-COM256 (default COM1)

baud = 9600, 19200, 38400, 57600, or 115200 (default 9600 baud)

parity = N (no parity)

stop_bit = 1 (one stop bit)

handshake = NONE or RTSCTS (default NONE for no handshaking)

For example, COM1:9600,N,8,1,NONE.

Where *connection* is one of these for the C-SPY GDB Server driver:

Via Ethernet *TCPIP:ip_address*

TCPIP:ip_address,port

TCPIP:hostname

TCPIP:hostname,port

Note that if no port is specified, port 3333 is used by default.

Where *connection* is one of these for the C-SPY IAR ROM-monitor driver:

Via serial port *port:baud,parity,stop_bit,handshake*

port = COM1-COM256 (default COM1)

baud = 9600, 19200, 38400, 57600, or 115200 (default 9600 baud)

parity = N (no parity)

stop_bit = 1 (one stop bit)

handshake = NONE or RTSCTS (default NONE for no handshaking)

For example, COM1:9600,N,8,1,NONE.

Where *connection* is one of these for the C-SPY J-Link/J-Trace driver:

Via USB port USB:#*serial* where *serial* is a string of digits and letters that identifies which probe you want to connect to. The serial number can be found either printed on the probe or obtained by connecting only one probe and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box.

USB:#*select* forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.

Via USB directly to the debug probe USB0-USB3

When using USB0 and if there are more than one debug probes on the USB connection, a dialog box is displayed when the debug session starts. Use the dialog box to choose which debug probe to connect to.

Via J-Link on LAN TCPIP:

When the colon sign is not followed by any address, host name, or serial number, the J-Link driver searches for all J-Link debug probes on the local network and displays them in a dialog box where you can choose which one to connect to (Auto detect).

TCPIP:*ip_address*

TCPIP:*ip_address,port*

TCPIP:*hostname*

TCPIP:*hostname,port*

TCPIP:#*serial*, connects to the J-Link with the serial number *number* on the local network

Note that if no port is specified, port 19020 is used by default.

Where *connection* is one of these for the C-SPY I-jet/JTAGjet driver:

Via USB port `USB:#serial` where *serial* is a string of digits and letters that identifies which probe you want to connect to. The serial number can be found either printed on the probe or obtained by connecting only one probe and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box.

`USB:#select` forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.

Where *connection* is one of these for the C-SPY Macraigor driver:

For mpDemon `port:baud`
 `port = COM1-COM4`
 `baud = 9600, 19200, 38400, 57600, or 115200 (default 9600 baud)`

For mpDemon `TCPIP:ip_address`
 `TCPIP:ip_address,port`
 `TCPIP:hostname`
 `TCPIP:hostname,port`
 Note that if no port is specified, port 19020 is used by default.

Via USB to usbDemon and usb2Demon `USB ports = USB0-USB3`

Where *connection* is one of these for the C-SPY ST-LINK driver:

Via USB port

USB:#*serial* where *serial* is a string of digits and letters that identifies which probe you want to connect to. The serial number can be found either printed on the probe or obtained by connecting only one probe and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box.

USB:#*select* forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.

USB*x* where *x* is the enumeration order (0-256) of the probe when plugged in. This is an alternative notation for when the serial number cannot be used—a solution for older probes. However, this is an uncertain method, because the order can change the next time that you plug in the probes, or when you reboot your computer. The USB port can be obtained by plugging in all probes to be used. Then use `--drv_communication=USB:#select` to display all connected probes in the **Debug Probe Selection** dialog box.

Where *connection* is one of these for the C-SPY TI Stellaris driver:

Via USB port

USB:#*serial* where *serial* is a string of digits and letters that identifies which probe you want to connect to. The serial number can be found either printed on the probe or obtained by connecting only one probe and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box.

USB:#*select* forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.

USB*x* where *x* is the enumeration order (0-256) of the probe when plugged in. This is an alternative notation for when the serial number cannot be used—a solution for older probes. However, this is an uncertain method, because the order can change the next time that you plug in the probes, or when you reboot your computer. The USB port can be obtained by plugging in all probes to be used. Then use `--drv_communication=USB:#select` to display all connected probes in the **Debug Probe Selection** dialog box.

Where *connection* is one of these for the C-SPY TI XDS driver:

Via USB port
(XDS100 and
XDS110)

USB:#*serial* where *serial* is a string of digits and letters that identifies which probe you want to connect to. The serial number can be found either printed on the probe or obtained by connecting only one probe and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box.

USB:#*select* forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.

USB*x* where *x* is the enumeration order (0-256) of the probe when plugged in. This is an alternative notation for when the serial number cannot be used—a solution for older probes. However, this is an uncertain method, because the order can change the next time that you plug in the probes, or when you reboot your computer. The USB port can be obtained by plugging in all probes to be used. Then use `--drv_communication=USB:#select` to display all connected probes in the **Debug Probe Selection** dialog box.

Via COM port
(XDS200 only)

COM*x* where *x* is the enumeration order (0-256) of the probe when plugged in. This is an uncertain method, because the order can change the next time that you plug in the probes, or when you reboot your computer.

COM:#*select* forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.

For use with

The C-SPY Angel debug monitor driver
The C-SPY GDB Server driver
The C-SPY IAR ROM-monitor driver
The C-SPY J-Link/J-Trace driver
The C-SPY Macraigor driver
The C-SPY ST-LINK driver
The C-SPY TI Stellaris driver
The C-SPY TI XDS driver.

Description

Use this option to choose communication link.



Project>Options>Debugger>Angel>Communication

Project>Options>Debugger>GDB Server>TCP/IP address or hostname [,port]

Project>Options>Debugger>J-Link/J-Trace>Connection>Communication

To set related options for the C-SPY Macraigor driver, choose:

Project>Options>Debugger>Macraigor

To set this option for the C-SPY ST-LINK driver, the C-SPY TI Stellaris driver, and the C-SPY TI XDS driver, use **Project>Options>Debugger>Extra Options**.

--drv_communication_log

Syntax	<code>--drv_communication_log=<i>filename</i></code>	
Parameters	<code>filename</code>	The name of the log file.
For use with	All C-SPY hardware drivers.	
Description	Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the communication protocol is required.	



Project>Options>Debugger>Driver>Log communication

--drv_default_breakpoint

Syntax	<code>--drv_default_breakpoint={0 1 2}</code>	
Parameters	0	Auto (default)
	1	Hardware
	2	Software
For use with	The C-SPY GDB Server driver The C-SPY I-jet/JTAGjet driver The C-SPY J-Link/J-Trace driver The C-SPY CMSIS-DAP driver	

	The C-SPY Macraigor driver
	The C-SPY TI XDS driver.
Description	Use this option to select the type of breakpoint resource to be used when setting a breakpoint.
See also	<i>Breakpoints options dialog box</i> , page 153.



Project>Options>Debugger>Driver>Breakpoints>Default breakpoint type

--drv_interface

Syntax	<code>--drv_interface={SWD JTAG}</code>				
Parameters	<table> <tr> <td>SWD</td> <td>Specifies the SWD interface.</td> </tr> <tr> <td>JTAG (default)</td> <td>Specifies the JTAG interface</td> </tr> </table>	SWD	Specifies the SWD interface.	JTAG (default)	Specifies the JTAG interface
SWD	Specifies the SWD interface.				
JTAG (default)	Specifies the JTAG interface				
For use with	<p>The C-SPY CMSIS-DAP driver</p> <p>The C-SPY I-jet/JTAGjet driver</p> <p>The C-SPY J-Link/J-Trace driver</p> <p>The C-SPY Macraigor driver</p> <p>The C-SPY ST-LINK driver.</p> <p>The C-SPY TI Stellaris driver</p> <p>The C-SPY TI XDS driver.</p>				
Description	<p>Use this option to specify the communication interface between the debug probe and the target system.</p> <p>The SWD interface uses fewer pins than JTAG. Specify <code>--drv_interface=SWD</code> if you want to use the serial-wire output (SWO) communication channel. Alternatively, you can set this option to <code>JTAG</code> and also specify the <code>--jet_swo_on_d0</code> option. SWO output on <code>Trace_D0</code> is only supported by the C-SPY I-Jet/I-jet Trace driver.</p> <p>Note that if you select stdout/stderr via SWO on the General Options>Library Configuration page, SWD is selected automatically, unless the device supports output of SWO on <code>Trace_D0</code>.</p>				

Project>Options>Debugger>Macraigor>JTAG speed

Project>Options>Debugger>ST-LINK>Setup>JTAG/SWD speed

Project>Options>Debugger>TI Stellaris>Setup>JTAG/SWD speed

Project>Options>Debugger>TI XDS>Setup>JTAG/SWD speed

--drv_reset_to_cpu_start

Syntax `--drv_reset_to_cpu_start`

For use with

- The C-SPY Angel debug monitor driver
- The C-SPY GDB Server driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY TI Stellaris driver
- The C-SPY TI XDS driver
- The C-SPY Macraigor driver
- The C-SPY RDI driver
- The C-SPY ST-LINK driver
- The C-SPY TI Stellaris driver
- The C-SPY TI XDS driver.

Description

Normally, at reset, the debugger sets PC to the entry point of the application.

This option omits setting the PC each time that the application is reset. This can be useful when you want to keep the reset value that the CPU sets at reset, for example to start executing from the very first instruction pointed out by the vector table, or to run a bootloader or OS startup code before entering the start address of the application.


This option also keeps the value of the SP (for Cortex-M) or CPSR register (for other devices) set by the CPU.



To set this option, use **Project>Options>Debugger>Extra Options**.

--drv_restore_breakpoints

Syntax `--drv_restore_breakpoints=location`

ParametersParameters	<i>location</i>	Address or function name label
For use with	The C-SPY CMSIS-DAP driver The C-SPY GDB Server driver The C-SPY I-jet/JTAGjet driver The C-SPY J-Link/J-Trace driver The C-SPY CMSIS-DAP driver The C-SPY Macraigor driver The C-SPY ST-LINK driver The C-SPY TI XDS driver.	
Description	Use this option to restore automatically any software breakpoints that were overwritten during system startup.	
See alsoSee also	<i>Breakpoints options dialog box</i> , page 153.	
		Project>Options>Debugger>Driver>Breakpoints>Restore software breakpoints at

--drv_swo_clock_setup

Syntax	<code>--drv_swo_clock_setup=<i>frequency, autodetect, wanted</i></code>	
Parameters	<i>frequency</i>	The exact clock frequency used by the internal processor clock, <code>HCLK</code> , in Hz. This value is used for configuring the SWO communication speed and for calculating timestamps.
	<i>autodetect</i>	0, Specify the wanted frequency using the parameter <i>wanted</i> . 1, Automatically uses the highest possible frequency that the J-Link debug probe can handle.
	<i>wanted</i>	The frequency to be used, if <i>autodetect</i> is 0, in Hz. Use <i>wanted</i> if data packets are lost during transmission.
For use with	The C-SPY J-Link/J-Trace driver	

Description The C-SPY ST-LINK driver.

Use this option to set up the CPU clock. If this option is not used, the CPU clock frequency is by default set to 72 MHz.



Project>Options>Debugger>J-Link/J-Trace>Setup>CPU clock
Project>Options>Debugger>J-Link/J-Trace>Setup>SWO clock

--drv_vector_table_base

Syntax `--drv_vector_table_base=expression`

Parameters *expression* A label or an address

For use with


- The C-SPY GDB Server driver
- The C-SPY I-jet/JTAGjet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY TI Stellaris driver
- The C-SPY TI XDS driver
- The C-SPY Macraigor driver
- The C-SPY RDI driver
- The C-SPY ST-LINK driver
- The C-SPY Simulator driver.

Description Use this option to specify the location of the reset vector (this also determines the placement of the initial stack pointer value for Cortex-M). This is useful if you want to override the default `__vector_table` label—defined in the system startup code—in the application or if the application lacks this label, which can be the case if you debug code that is built by tools from another vendor.




To set this option, use **Project>Options>Debugger>Extra Options**.

-f

Syntax	<code>-f filename</code>
Parameters	<i>filename</i> A text file that contains the commands (default filename extension <code>.xml</code>).
For use with	<code>cspybat</code> This option can be placed both before and after the <code>--backend</code> option on the command line.
Description	Use this option to make <code>cspybat</code> read command line options from the named file. In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character is treated like a space or tab character. Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.
	 To set this option, use Project>Options>Debugger>Extra Options .

--flash_loader

Syntax	<code>--flash_loader filename</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The flash loader specification XML file, with the filename extension <code>.board</code> .
For use with	<code>cspybat</code>
Description	Use this option to specify a flash loader specification xml file which contains all relevant information about the flash loading. There can be more than one such argument, in which case each argument will be processed in the specified order, resulting in several flash programming passes.
See also	The <i>IAR Embedded Workbench flash loader User Guide</i> .
	 To set related options, choose: Project>Options>Debugger>Use flash loader(s)

--gdbserv_exec_command


Syntax	<code>--gdbserv_exec_command="string"</code>	
Parameters	<code>"string"</code>	String or command sent to the GDB Server; see its documentation for more information.
For use with	The C-SPY GDB Server driver.	
Description	Use this option to send strings or commands to the GDB Server.	

**Project>Options>Debugger>Extra Options****--jet_board_cfg**

Syntax	<code>--jet_board_cfg=probe_configuration_file</code>	
Parameters	<code>probe_configuration_file</code>	The full path to a probe configuration file.
For use with	The C-SPY I-jet/JTAGjet driver The C-SPY CMSIS-DAP driver	
Description	Use this option to specify a probe configuration file that defines the debug system on the board.	

**Project>Options>Debugger>CMSIS DAP>JTAG/SWD>Probe configuration file****Project>Options>Debugger>I-jet/JTAGjet>JTAG/SWD>Probe configuration file****--jet_board_did**

Syntax	<code>--jet_board_did={cpu #cpu_number}</code>	
Parameters	<code>cpu</code>	If a board configuration file is specified (using <code>--jet_board_cfg</code>) and the defined debug system contains more than one CPU, use this parameter to select a CPU. The value of <code>cpu</code> is a text string. The range of valid values can be found in the probe configuration file.

	<i>#cpu_number</i>	<p>If the debug system is a multi-core SWD system, specify the CPU number on the DAP.</p> <p>If the debug system is a JTAG scan chain, and there are several CPUs at the specified TAP position, then specify the CPU number on target.</p> <p>Note that <i>#cpu_number</i> has no effect if a board configuration file is specified using <code>--jet_board_cfg</code>.</p>
For use with	<p>The C-SPY CMSIS-DAP driver</p> <p>The C-SPY I-jet/JTAGjet driver</p>	
Description	<p>Use this option to specify which CPU to debug on a multi-core system.</p> <p><code>--jet_board_did=#cpu_number</code> is applicable also when <code>--jet_probe=cmsisdap</code> is specified.</p>	
Example	<p>Selecting the CPU on a multi-core device with a probe configuration file:</p> <pre>--jet-board-cfg=device.ProbeConfig --jet_board_did=A9_1</pre> <p>Selecting the CPU on a multi-core device with a JTAG scan chain, where several CPUs are found at the specified TAP position:</p> <pre>--jet_tap_position=1 --jet_ir_length=5 --jet_board_did=#2</pre> <p> Project>Options>Debugger>CMSIS DAP>JTAG/SWD>Probe configuration file>CPU</p> <p>Project>Options>Debugger>CMSIS DAP/JTAG/SWD>Probe config>Explicit>CPU number on target</p> <p>Project>Options>Debugger>I-jet/JTAGjet>JTAG/SWD>Probe configuration file>CPU</p> <p>Project>Options>Debugger>I-jet/JTAGjet/JTAG/SWD>Explicit probe configuration>CPU number on target</p>	

--jet_cpu_clock

Syntax	<code>--jet_cpu_clock=frequency</code>	
Parameters	<i>frequency</i>	The clock frequency in Hz

For use with	The C-SPY I-jet/JTAGjet driver.
Description	Use this option to specify the exact clock frequency used by the internal processor clock, HCLK. This value is used for configuring the SWO communication speed and for calculating timestamps. Note: This option is relevant only when the option <code>--jet_swo_protocol</code> is set to UART.



Project>Options>Debugger>I-jet/JTAGjet>Trace>SWO clock setup>CPU clock

--jet_ir_length

Syntax	<code>--jet_ir_length=length</code>
Parameters	<i>length</i> The number of IR bits preceding the ARM core to connect to, for JTAG scan chains that mix ARM devices with other devices.
For use with	The C-SPY I-jet/JTAGjet driver.
Description	Use this option to set the number of IR bits preceding the ARM core to connect to.
See also	<i>JTAG/SWD options for I-jet/JTAGjet</i> , page 517



Project>Options>Debugger>I-jet/JTAGjet>JTAG/SWD>Explicit probe configuration>Preceding bits

--jet_power_from_probe

Syntax	<code>--jet_power_from_probe=[leave_on switch_off]</code>
Parameters	<i>leave_on</i> Continues to supply power to the target even after the debug session has been stopped. <i>switch_off</i> Turns off the power to the target when the debug session stops.
For use with	The C-SPY I-jet/JTAGjet driver.

Description Use this option to specify the status of the probe power supply after debugging. If this option is not specified, the probe will not supply power to the board.



Project>Options>Debugger>I-jet/JTAGjet>Setup>Target power

--jet_probe

Syntax `--jet_probe=[i jet | cmsisdap]`

Parameters

<code>i jet</code>	Specifies the C-SPY I-jet/JTAGjet driver as the interface to an I-jet, I-jet Trace, or JTAGjet probe.
<code>cmsisdap</code>	Specifies the C-SPY I-jet/JTAGjet driver as the interface to a CMSIS-DAP system.

For use with The C-SPY I-jet/JTAGjet driver
The C-SPY CMSIS-DAP driver

Description Use this option to specify the C-SPY I-jet/JTAGjet driver as the interface to a debug system.



Project>Options>Debugger>Driver

--jet_script_file

Syntax `--jet_script_file=path`

Parameters

<code>path</code>	The path to the file where the scripted reset strategies are described.
-------------------	---

For use with The C-SPY I-jet/JTAGjet driver
The C-SPY CMSIS-DAP driver

Description Use this option to specify the file that describes the available scripted reset strategies, if any.

See also `--reset_style`, page 483 and `--jet_standard_reset`, page 470.



To set this option, use **Project>Options>Debugger>Extra Options**.

--jet_standard_reset

Syntax

--jet_standard_reset=*strategy*,*duration*,*delay*

Parameters

strategy

The reset strategy. Choose between:

- 0, reset disabled
- 1, software reset
- 2, hardware reset
- 3, core reset
- 4, system reset.

The following reset strategies are available, if present in the file specified by --jet_script_file and defined by corresponding instances of --reset_style:


- 5, custom reset
- 6, reset by watchdog or reset register
- 7, reset and halt after bootloader
- 8, reset and halt before bootloader
- 9, connect during reset

duration


The time in milliseconds that the hardware reset asserts the reset signal (line nSRST/nRESET) low to reset the device.

Some devices might require a longer reset signal than the default 200 ms.

This parameter applies to the hardware reset, and to those custom reset strategies that use the hardware reset.

	<i>delay</i>	<p>The delay time, in milliseconds, after the reset signal has been de-asserted, before the debugger attempts to control the processor.</p> <p>The processor might be kept internally in reset for some time after the external reset signal has been de-asserted, thus inaccessible for the debugger.</p> <p>This parameter applies to the Hardware reset, and to those custom reset strategies that use the Hardware reset.</p>
For use with	<p>The C-SPY CMSIS-DAP driver</p> <p>The C-SPY I-jet/JTAGjet driver</p>	
Description	<p>Use this option to select the reset strategy to be used when the debugger starts. Note that Cortex-M uses a different set of strategies than other devices.</p>	
See also	<p><i>--reset_style</i>, page 483 and <i>--jet_script_file</i>, page 469.</p>	
		<p>Project>Options>Debugger>CMSIS DAP>Setup>Reset</p> <p>Project>Options>Debugger>I-jet/JTAGjet>Setup>Reset</p>

--jet_startup_connection_timeout

Syntax	<i>--jet_startup_connection_timeout=milliseconds</i>	
Parameters	<i>milliseconds</i>	The time in milliseconds.
For use with	<p>The C-SPY CMSIS-DAP driver</p> <p>The C-SPY I-jet/JTAGjet driver</p>	
Description	<p>Use this option to prolong the time that the C-SPY driver tries to connect to the target board.</p>	
		To set this option, use Project>Options>Debugger>Extra Options .

--jet_swo_on_d0

Syntax	<i>--jet_swo_on_d0</i>
--------	------------------------

For use with	The C-SPY I-jet/JTAGjet driver.
Description	Use this option to specify that SWO trace data is output on the trace data pin D0. When using this option, both the SWD and the JTAG interface can handle SWO trace data.



Project>Options>Debugger>I-jet/JTAGjet>SWO>SWO on the TraceD0 pin

--jet_swo_prescaler

Syntax	<code>--jet_swo_prescaler=<i>number</i></code>	
Parameters	<i>number</i>	The prescaler value, 1–100, which in turn determines the CPU clock frequency.
For use with	The C-SPY I-jet/JTAGjet driver.	
Description	Use this option to specify the prescaler for the SWO clock. The CPU clock frequency is divided by the number specified as the prescaler. If data packets are lost during transmission, try using a higher prescaler value.	

If this option is not specified, a prescaler value is set automatically. This automatically set value is the highest possible frequency that the debug probe can handle.



I-jet/JTAGjet>SWO clock setup>SWO prescaler

--jet_swo_protocol

Syntax	<code>--jet_swo_protocol={<i>auto</i> <i>Manchester</i> <i>UART</i>}</code>	
Parameters	<i>auto</i>	Automatically selects the communication protocol.
	<i>Manchester</i>	Specifies the Manchester protocol.
	<i>UART</i>	Specifies the UART protocol.
For use with	The C-SPY I-jet/JTAGjet driver.	
Description	Use this option to specify the communication protocol for the SWO channel. If this option is not specified, <i>auto</i> is automatically used.	

**Project>Options>Debugger>I-jet/JTAGjet>SWO protocol**

--jet_tap_position

Syntax	<code>--jet_tap_position=tap_number multidrop_id</code>	
Parameters	<i>tap_number</i>	The TAP position of the device you want to connect to.
	<i>multidrop_id</i>	The target ID in a multi-drop system.
For use with	The C-SPY I-jet/JTAGjet driver The C-SPY CMSIS-DAP driver	
Description	If you are using the JTAG interface, and there is more than one device on the JTAG scan chain, use this option to select a specific device. If you are using the SWD interface, and there is a multi-drop SWD system on the board, use this option to select a target ID.	
See also	<i>JTAG/SWD options for I-jet/JTAGjet</i> , page 517.	


**Project>Options>Debugger>I-jet/JTAGjet>JTAG/SWD>Explicit probe configuration>Target number (TAP or Multidrop ID)**

--jlink_dcc_timeout


Syntax	<code>--jlink_dcc_timeout=milliseconds</code>	
Parameters	<i>milliseconds</i>	The timeout in milliseconds. The valid range is 5-5000. The default value is 100 milliseconds.
For use with	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to specify a timeout for a pending request from C-SPY to the DCC agent on target.	

To set this option, use **Project>Options>Debugger>Extra Options**.

--jlink_device_select


Syntax	<code>--jlink_device_select=tap_number</code>
Parameters	<i>tap_number</i> The TAP position of the device you want to connect to.
For use with	The C-SPY J-Link/J-Trace driver.
Description	If there is more than one device on the JTAG scan chain, use this option to select a specific device.
See also	<i>JTAG/SWD options for I-jet/JTAGjet</i> , page 517.
	 Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>TAP number

--jlink_exec_command


Syntax	<code>--jlink_exec_command=cmdstr1; cmdstr2; cmdstr3 ...</code>
Parameters	<i>cmdstrn</i> J-Link/J-Trace command string.
For use with	The C-SPY J-Link/J-Trace driver.
Description	Use this option to make the debugger call the <code>__jlinkExecCommand</code> macro with one or several command strings, after target connection has been established.
See also	<i>__jlinkExecCommand</i> , page 398.
	 To set this option, use Project>Options>Debugger>Extra Options .

--jlink_initial_speed

Syntax	<code>--jlink_initial_speed=speed</code>
Parameters	<i>speed</i> The initial communication speed in kHz. If no speed is specified, 32 kHz will be used as the initial speed.

For use with	The C-SPY J-Link/J-Trace driver.
Description	Use this option to set the initial JTAG communication speed in kHz.
See also	<i>Setup options for J-Link/J-Trace</i> , page 523.
	 Project>Options>Debugger>J-Link/J-Trace>Setup>JTAG speed>Fixed

--jlink_ir_length

Syntax	<code>--jlink_ir_length=length</code>	
Parameters	<i>length</i>	The number of IR bits preceding the ARM core to connect to, for JTAG scan chains that mix ARM devices with other devices.
For use with	The C-SPY J-Link/J-Trace driver.	
Description	Use this option to set the number of IR bits preceding the ARM core to connect to.	
See also	<i>Connection options for J-Link/J-Trace</i> , page 528.	
	 Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>Preceding bits	

--jlink_reset_strategy

Syntax	<code>--jlink_reset_strategy=delay, strategy</code>	
Parameters	<i>delay</i>	For Cortex-M and ARM 7/9/11 with strategies 1-9, <i>delay</i> should be 0 (ignored). For ARM 7/9/11 with strategy 0, the delay should be one of 0-10000.
	<i>strategy</i>	For information about supported reset strategies, see the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> .
For use with	The C-SPY J-Link/J-Trace driver.	

Description Use this option to select the reset strategy to be used at debugger startup.

See also *Setup options for J-Link/J-Trace*, page 523.



Project>Options>Debugger>J-Link/J-Trace>Setup>Reset

--jlink_script_file

Syntax `--jlink_script_file=filename`

Parameters

<i>filename</i>	The name of the J-Link script file.
-----------------	-------------------------------------

For use with The C-SPY J-Link/J-Trace driver.

Description Use this option to specify the J-Link script file to be used.
 J-Link has a script language that can be used for setting up hardware. For certain targets, ready-made script files are automatically pointed out by IAR Embedded Workbench. In command line mode, the script file needs to be manually specified by using this option.

See also *The J-Link/J-Trace ARM User Guide* (JLinkARM.pdf, document number UM08001), section 5.10, for a detailed description of the script language.



To set this option using a non-predefined script file, use **Project>Options>Debugger>Extra Options**.

--jlink_trace_source

Syntax `--jlink_trace_source={ETB|ETM}`

Parameters

ETB	Selects ETB trace.
ETM	Selects ETM trace.

For use with The C-SPY J-Link/J-Trace driver.

Description Use this option to select either ETB or ETM as the trace source.
Note: This option applies only to J-Trace.

See also

Setup options for J-Link/J-Trace, page 523.



Project>Options>Debugger>J-Link/J-Trace>Setup>ETM/ETB

--leave_running

Syntax

`--leave_running`

Note that this option must be placed before the `--backend` option on the command line.

For use with

`cspybat`

Description

Makes `cspybat` start the execution on the target and then exits but leaves the target running.



To set a related option, choose:

Project>Options>Debugger>Attach to running target

--macro

Syntax

`--macro filename`

Note that this option must be placed before the `--backend` option on the command line.

Parameters

filename

The C-SPY macro file to be used (filename extension `mac`).

For use with

`cspybat`

Description

Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.

See also

Briefly about using C-SPY macros, page 366.



Project>Options>Debugger>Setup>Setup macros>Use macro file

--macro_param

Syntax

`--macro_param [param=value]`

Note that this option must be placed before the `--backend` option on the command line.

Parameters	<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 <i>value</i> 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 373.



Project>Options>Debugger>Extra Options

--mac_handler_address

Syntax	<code>--mac_handler_address=<i>address</i></code>
Parameters	<i>address</i> The start address of the memory area for the debug handler.
For use with	The C-SPY Macraigor driver.
Description	Use this option to specify the location—the memory address—of the debug handler used by Intel XScale devices.
See also	<i>Macraigor</i> , page 530.



Project>Options>Debugger>Macraigor>Debug handler address

--mac_jtag_device

Syntax	<code>--mac_jtag_device=<i>device</i></code>
Parameters	<i>device</i> The device corresponding to the hardware interface that is used. Choose between Macraigor <code>mpDemon</code> , <code>usbDemon</code> , and <code>usb2demon</code> .
For use with	The C-SPY Macraigor driver.

Description Use this option to select the device corresponding to the hardware interface that is used.

See also *Macraigor*, page 530.



Project>Options>Debugger>Macraigor>OCD interface device

--mac_multiple_targets

Syntax `--mac_multiple_targets=<tap-no>@dev0, dev1, dev2, dev3, . . .`

Parameters

tap-no The TAP number of the device to connect to, where 0 connects to the first device, 1 to the second, and so on.

dev0-devn The nearest TDO pin on the Macraigor JTAG probe.

For use with The C-SPY Macraigor driver.

Description If there is more than one device on the JTAG scan chain, each device must be defined. Use this option to specify which device you want to connect to.

Example `--mac_multiple_targets=0@ARM7TDMI, ARM7TDMI`

See also *Macraigor*, page 530.



Project>Options>Debugger>Macraigor>JTAG scan chain with multiple targets

--mac_reset_pulls_reset

Syntax `--mac_reset_pulls_reset=time`

Parameters

time 0–2000 which is the delay in milliseconds after reset.

For use with The C-SPY Macraigor driver.

Description Use this option to make C-SPY perform an initial hardware reset when the debugger is started, and to specify the delay for the reset.

See also *Macraigor*, page 530.

**Project>Options>Debugger>Macraigor>Hardware reset****--mac_set_temp_reg_buffer**

Syntax	<code>--mac_set_temp_reg_buffer=<i>address</i></code>	
Parameters	<i>address</i>	The start address of the RAM area.
For use with	The C-SPY Macraigor driver.	
Description	Use this option to specify the start address of the RAM area that is used for controlling the MMU and caching via the CP15 coprocessor.	

To set this option, use **Project>Options>Debugger>Extra Options**.**--mac_xscale_ir7**

Syntax	<code>--mac_xscale_ir7</code>	
For use with	The C-SPY Macraigor driver.	
Description	Use this option to specify that the XScale ir7 core is used, instead of XScale ir5. Note that this option is mandatory when using the XScale ir7 core.	
	These XScale cores are supported by the C-SPY Macraigor driver:	
	Intel XScale Core 1 (5-bit instruction register—ir5)	
	Intel XScale Core 2 (7-bit instruction register—ir7)	

To set this option, use **Project>Options>Debugger>Extra Options**.**--mapu**

Syntax	<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 `stderr` and the execution will stop.

See also *Memory access checking*, page 164.



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 51.



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 `armbat.dll` located in the `arm\bin` directory must be used.

Use this option to include this plugin during the debug session. This option can be used more than once on the command line.

Note: You can use this option to include also other plugin modules, but in that case the module must be able to work with `cspybat` specifically. This means that the C-SPY plugin modules located in the `common\plugin` directory cannot normally be used with `cspybat`.



Project>Options>Debugger>Plugins

--proc_stack_stack

Syntax	<code>--proc_stack_stack=startaddress, endaddress</code>	
	where <i>stack</i> is one of <code>main</code> or <code>proc</code> for Cortex-M and	
	where <i>stack</i> is one of <code>usr</code> , <code>svc</code> , <code>irq</code> , <code>fig</code> , <code>und</code> , or <code>abt</code> for other ARM cores	
Parameters	<i>startaddress</i>	The start address of the stack, specified either as a value or as an expression.
	<i>endaddress</i>	The end address of the stack, specified either as a value or as an expression.
For use with	All C-SPY drivers. Note that this command line option is only available when using C-SPY from the IDE; not in batch mode using <code>cspybat</code> .	
Description	Use this option to provide C-SPY with information about reserved stacks. By default, C-SPY receives this information from the system startup code, but if you for some reason want to override the default values, this option can be useful.	
Example	<code>--proc_stack_irq=0x8000,0x80FF</code>	



To set this option, use **Project>Options>Debugger>Extra Options**.

--rdi_allow_hardware_reset

Syntax	<code>--rdi_allow_hardware_reset</code>
For use with	The C-SPY RDI driver.

Description Use this option to allow the emulator to perform a hardware reset of the target. Requires support by the emulator.

See also *RDI*, page 532.



Project>Options>Debugger>RDI>Allow hardware reset

--rdi_driver_dll

Syntax `--rdi_driver_dll filename`

Parameters *filename* The file or path to the driver DLL file.

For use with The C-SPY RDI driver

Description Use this option to specify the path to the driver DLL file provided with the JTAG pod.

See also *RDI*, page 532.



Project>Options>Debugger>RDI>Manufacturer RDI driver

For JTAGjet, this option is not available in the IDE.

--rdi_step_max_one

Syntax `--rdi_step_max_one`

For use with The C-SPY Angel debug monitor driver
The C-SPY RDI driver.

Description Use this option to execute only one instruction. The debugger will turn off interrupts while stepping and, if necessary, simulate the instruction instead of executing it.



To set this option, use **Project>Options>Debugger>Extra Options**.

--reset_style

Syntax `--reset_style="reset_id, reset_name, selected, menu_command"`


Parameters	<p><i>reset_id</i> The number of the reset strategy, 0-9, as described for <code>--jet_standard_reset</code></p> <p><i>reset_name</i> The name of the reset strategy, according to the file specified by <code>--jet_script_file</code>.</p> <p>For the built-in reset strategies, this parameter is <code>-</code>. To override a built-in reset strategy, enter the label or function name in your reset script file.</p> <p><i>selected</i> 0 or 1, where 1 sets the default reset strategy for the Reset drop-down button</p> <p><i>menu_command</i> The name of the reset strategy as it will be displayed on the Reset drop-down menu.</p>
For use with	<p>The C-SPY CMSIS-DAP driver</p> <p>The C-SPY I-jet/JTAGjet driver</p>
Description	Use this option to specify the reset strategies that will be available when debugging, once for each reset strategy.
Example	<p>This example specifies a script file, sets the standard reset strategy, and specifies the reset strategies that will be available when debugging:</p> <pre>--jet_script_file=myDir\myProbeScriptFile --jet_standard_reset=9,0,0 --reset_style="0,-,0,Disabled (no reset)" --reset_style="1,-,0,Software" --reset_style="2,-,0,Hardware" --reset_style="3,-,0,Core" --reset_style="4,-,0,System" --reset_style="5,Custom,0,Custom reset" --reset_style="9,ConnectUnderReset,1,Connect during reset"</pre>
See also	<code>--jet_script_file</code> , page 469 and <code>--jet_standard_reset</code> , page 470




To set this option, use **Project>Options>Debugger>Extra Options**.

--semihosting

Syntax `--semihosting={none|iar_breakpoint}`

Parameters	No parameter	Use standard semihosting.
	none	Does not use semihosted I/O.
	iar_breakpoint	Uses the IAR proprietary semihosting variant.
For use with	All C-SPY drivers.	
Description	<p>Use this option to enable semihosted I/O and to choose the kind of semihosting interface to use. Note that if this option is not used, semihosting will by default be enabled and C-SPY will try to choose the correct semihosting mode automatically. This means that normally you do not have to use this option if your application is linked with semihosting.</p> <p>To make semihosting work, your application must be linked with a semihosting library.</p>	
See also	<p>The <i>IAR C/C++ Development Guide for ARM</i> for more information about linking with semihosting.</p> <p> Project>Options>General Options>Library Configuration</p>	

--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	cspybat	
Description	Use this option to omit the sign-on message.	
		This option is not available in the IDE.

--stlink_reset_strategy

Syntax	<code>--stlink_reset_strategy=delay, strategy</code>	
Parameters	<i>delay</i>	The delay time measured in milliseconds. <i>delay</i> is ignored and should be 0.

	<i>strategy</i>	The reset strategy. 0, (Normal) performs the standard reset procedure. 1, (Reset Pin) uses the reset pin to perform a hardware reset. Only available for ST-LINK version 2. 2, (Connect during reset) ST-LINK connects to the target while keeping Reset active (Reset is pulled low and remains low while connecting to the target). Only available for ST-LINK version 2.
For use with	The C-SPY ST-LINK driver.	
Description	Use this option to select the reset strategy to be used at debugger startup.	
See also	<i>Setup options for ST-LINK</i> , page 533	



Project>Options>Debugger>ST-LINK>Setup>Reset

--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.

--xds_board_file

Syntax	<code>--xds_board_file=dat_file</code>
Parameters	<i>dat_file</i> The (path and) filename of the board file.

For use with	The C-SPY TI XDS driver.
Description	Use this option to override the default board file by specifying a custom board file.  Project>Options>Debugger>TI XDS>Setup>Emulator>Specify custom board file Project>Options>Debugger>TI XDS>Setup>Emulator>Board file

--xds_reset_strategy

Syntax	<code>--stlink_reset_strategy=delay, strategy</code>				
Parameters	<table> <tr> <td><i>delay</i></td> <td>The delay time measured in milliseconds.</td> </tr> <tr> <td><i>strategy</i></td> <td> The reset strategy. For Cortex-M devices: 0, CPU reset 1, System Reset 2, Board reset. Only available for CC26xx and CC13xx devices. 3, Board reset, run and halt with delay. Available for all other Cortex-M devices. For other devices (not Cortex-M): 0, Software reset 1, Hardware reset </td> </tr> </table>	<i>delay</i>	The delay time measured in milliseconds.	<i>strategy</i>	The reset strategy. For Cortex-M devices: 0, CPU reset 1, System Reset 2, Board reset. Only available for CC26xx and CC13xx devices. 3, Board reset, run and halt with delay. Available for all other Cortex-M devices. For other devices (not Cortex-M): 0, Software reset 1, Hardware reset
<i>delay</i>	The delay time measured in milliseconds.				
<i>strategy</i>	The reset strategy. For Cortex-M devices: 0, CPU reset 1, System Reset 2, Board reset. Only available for CC26xx and CC13xx devices. 3, Board reset, run and halt with delay. Available for all other Cortex-M devices. For other devices (not Cortex-M): 0, Software reset 1, Hardware reset				


For use with	The C-SPY TI XDS driver.
Description	Use this option to select the reset strategy to be used at debugger startup.
See also	<i>Setup options for TI XDS</i> , page 537



Project>Options>Debugger>TI XDS>Setup>Reset

--xds_rootdir

Syntax	<code>--xds_rootdir=path</code>
--------	---------------------------------

For use with	The C-SPY TI XDS driver
Description	<p>Use this option to specify the path to the directory where the TI XDS driver package is installed. If you installed the package in an alternative location, you can use the global argument variable <code>XDS_EMUPACK_DIR</code> to set a new default value.</p> <p> To set this option, use Project>Options>Debugger>TI XDS>Setup>TI emulation package installation path.</p>

Flash loaders

- Introduction to the flash loader
- Using flash loaders
- Reference information on the flash loader

Introduction to the flash loader

A flash loader is an agent that is downloaded to the target. It fetches your application from the debugger and programs it into flash memory. The flash loader uses the file I/O mechanism to read the application program from the host. You can select one or several flash loaders, where each flash loader loads a selected part of your application. This means that you can use different flash loaders for loading different parts of your application.

Flash loaders for various microcontrollers is provided with IAR Embedded Workbench for ARM. In addition to these, more flash loaders are provided by chip manufacturers and third-party vendors. The flash loader API, documentation, and several implementation examples are available to make it possible for you to implement your own flash loader.

Using flash loaders

These tasks are covered:

- Setting up the flash loader(s)
- The flash loading mechanism
- Aborting a flash loader.

SETTING UP THE FLASH LOADER(S)

To use a flash loader for downloading your application:

- 1** Choose **Project>Options**.
- 2** Choose the **Debugger** category and click the **Download** tab.
- 3** Select the **Use Flash loader(s)** option. A default flash loader configured for the device you have specified will be used. The configuration is specified in a preconfigured `board` file.

- 4 To override the default flash loader or to modify the behavior of the default flash loader to suit your board, select the **Override default. board file** option, and **Edit** to open the **Flash Loader Configuration** dialog box. A copy of the *.board file will be created in your project directory and the path to the *.board file will be updated accordingly.
- 5 The **Flash Loader Overview** dialog box lists all currently configured flash loaders, see *Flash Loader Overview dialog box*, page 491. You can either select a flash loader or open the **Flash Loader Configuration** dialog box.

In the **Flash Loader Configuration** dialog box, you can configure the download. For more information about the various flash loader options, see *Flash Loader Configuration dialog box*, page 493.

THE FLASH LOADING MECHANISM

When the **Use flash loader(s)** option is selected and one or several flash loaders have been configured, these steps are performed when the debug session starts.

Steps 1 to 4 are performed for each flash loader in the flash loader configuration.

- 1 C-SPY downloads the flash loader into target RAM.
Steps 2 to 4 are performed one or more times depending on the size of the RAM and the size of the application image.
- 2 C-SPY writes code/data from the application image into target RAM (RAM buffer).
- 3 C-SPY starts execution of the flash loader.
- 4 The flash loader reads data from the RAM buffer and programs the flash memory.
- 5 The application image now resides in flash memory and can be started. The flash loader and the RAM buffer are no longer needed, so RAM is fully available to the application in the flash memory.

ABORTING A FLASH LOADER

To abort a flash loader:

- 1 Press Ctrl+Shift- (minus) for a short while.
- 2 A message that says that the flash loader has aborted is displayed in the Debug Log window.

This method can be used if you suspect that something is wrong with the execution, for example because it seems not to terminate in a reasonable time.

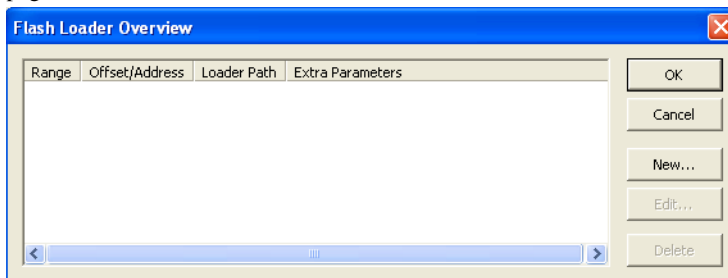
Reference information on the flash loader

Reference information about:

- *Flash Loader Overview dialog box*, page 491
- *Flash Loader Configuration dialog box*, page 493.

Flash Loader Overview dialog box

The **Flash Loader Overview** dialog box is available from the **Debugger>Download** page.



This dialog box lists all defined flash loaders. If you have selected a device on the **General Options>Target** page for which there is a flash loader, this flash loader is by default listed in the **Flash Loader Overview** dialog box.

The display area

Each row in the display area shows how you have set up one flash loader for flashing a specific part of memory:

Range

The part of your application to be programmed by the selected flash loader.

Offset/Address

The start of the memory where your application will be flashed. If the address is preceded with a, the address is absolute. Otherwise, it is a relative offset to the start of the memory.

Loader Path

The path to the flash loader *.flash file to be used (*.out for old-style flash loaders).

Extra Parameters

List of extra parameters that will be passed to the flash loader.

Click on the column headers to sort the list by range, offset/address, etc.

Function buttons

These function buttons are available:

OK

The selected flash loader(s) will be used for downloading your application to memory.

Cancel

Standard cancel.

New

Displays a dialog box where you can specify what flash loader to use, see *Flash Loader Configuration dialog box*, page 493.

Edit

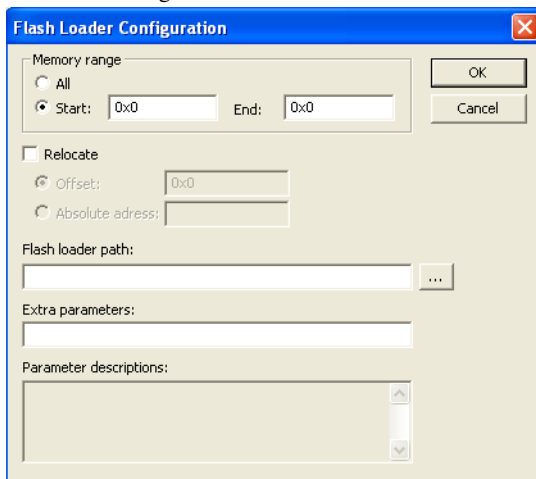
Displays a dialog box where you can modify the settings for the selected flash loader, see *Flash Loader Configuration dialog box*, page 493.

Delete

Deletes the selected flash loader configuration.

Flash Loader Configuration dialog box

The **Flash Loader Configuration** dialog box is available from the **Flash Loader Overview** dialog box.



Use the **Flash Loader Configuration** dialog box to configure the download to suit your board. A copy of the default `board` file will be created in your project directory.

Memory range

Specify the part of your application to be downloaded to flash memory. Choose between:

All

The whole application is downloaded using this flash loader.

Start/End

Specify the start and the end of the memory area for which part of the application will be downloaded.

Relocate

Overrides the default flash base address, in other words, relocates the location of the application in memory. This means that you can flash your application to a different location from where it was linked. Choose between:

Offset

A numeric value for a relative offset. This offset will be added to the addresses in the application file.

Absolute address

A numeric value for an absolute base address where the application will be flashed. The lowest address in the application will be placed on this address. Note that you can only use one flash loader for your application when you specify an absolute address.

You can use these numeric formats:

- 123456, decimal numbers
- 0x123456, hexadecimal numbers
- 0123456, octal numbers

The default base address used for writing the first byte—the lowest address—to flash is specified in the linker configuration file used for your application. However, it can sometimes be necessary to override the flash base address and start at a different location in the address space. This can, for example, be necessary for devices that remap the location of the flash memory.

Flash loader path

Use the text box to specify the path to the flash loader file (`*.flash`) to be used by your board configuration.

Extra parameters

Some flash loaders define their own set of specific options. Use this text box to specify options to control the flash loader. For information about available flash loader options, see the **Parameter descriptions** field.

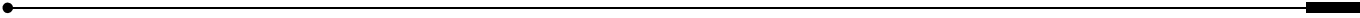
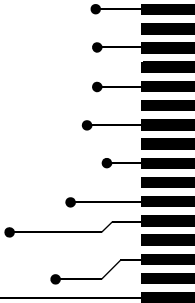
Parameter descriptions

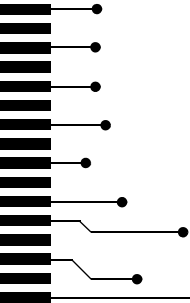
Displays a description of the extra parameters specified in the **Extra parameters** text box.

Part 4. Additional reference information

This part of the *C-SPY® Debugging Guide for ARM* 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 498.

- 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 Angel debug monitor driver	<i>Angel</i> , page 506
C-SPY GDB Server driver	<i>GDB Server</i> , page 512 <i>Breakpoints options dialog box</i> , page 153
C-SPY IAR ROM-monitor driver	<i>IAR ROM-monitor</i> , page 513
C-SPY CMSIS-DAP driver	<i>Setup options for CMSIS-DAP</i> , page 507 <i>JTAG/SWD options for CMSIS-DAP</i> , page 510 <i>Breakpoints options dialog box</i> , page 153
C-SPY I-jet/JTAGjet driver	<i>Setup options for I-jet/JTAGjet</i> , page 514 <i>JTAG/SWD options for I-jet/JTAGjet</i> , page 517 <i>Trace options for I-jet/JTAGjet</i> , page 519 <i>Breakpoints options dialog box</i> , page 153

Table 52: Options specific to the C-SPY drivers you are using

C-SPY driver	Available options pages
C-SPY J-Link/J-Trace driver	Setup options for J-Link/J-Trace, page 523 Connection options for J-Link/J-Trace, page 528 Breakpoints options dialog box, page 153
C-SPY TI Stellaris driver	Setup options for TI Stellaris, page 536
C-SPY TI XDS driver	Setup options for TI XDS, page 537
C-SPY Macraigor driver	Macraigor, page 530
RDI driver	RDI, page 532
ST-LINK driver	Setup options for ST-LINK, page 533
Third-party driver	Third-Party Driver options, page 539.

Table 52: Options specific to the C-SPY drivers you are using (Continued)

- 5** To restore all settings to the default factory settings, click the **Factory Settings** button.
- 6** When you have set all the required options, click **OK** in the **Options** dialog box.

Reference information on general debugger options

Reference information about:

- Setup
- Download
- Images
- Extra Options
- Plugins

Setup

The general **Setup** options select the C-SPY driver, the setup macro file, and device description file to use, and specify which default source code location to run to.

Driver

Selects the C-SPY driver for the target system you have.

Run to

Specifies the location C-SPY runs to when the debugger starts after a reset. By default, C-SPY runs to the `main` function.

To override the default location, specify the name of a different location you want C-SPY to run to. You can specify assembler labels or whatever can be evaluated as such, for example function names.

If the option is deselected, the program counter will contain the regular hardware reset address at each reset.

Setup macros

Registers the contents of a setup macro file in the C-SPY startup sequence. Select **Use macro file** and specify the path and name of the setup file, for example `SetupSimple.mac`. If no extension is specified, the extension `mac` is assumed. A browse button is available for your convenience.

It is possible to specify up to two different macro files.

Device description file

A default device description file—either an IAR-specific `ddf` file or a CMSIS System View Description file—is selected automatically based on your project settings. To

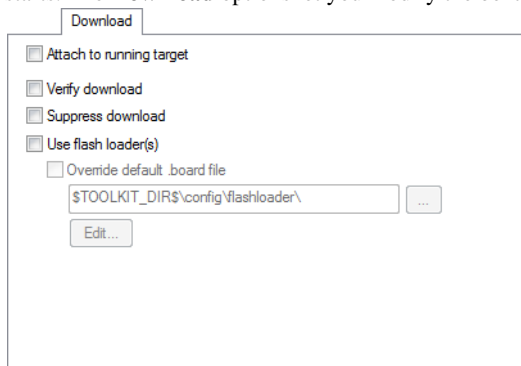
override the default file, select **Override default** and specify an alternative file. A browse button is available for your convenience.

For information about the device description file, see *Modifying a device description file*, page 54.

IAR-specific device description files for each arm device are provided in the directory `arm\config` and have the filename extension `.ddf`.

Download

By default, C-SPY downloads the application to RAM or flash when a debug session starts. The **Download** options let you modify the behavior of the download.



Attach to running target

Makes the debugger attach to a running application at its current location, without resetting or halting (for J-Link and I-jet/JTAGjet only) the target system. To avoid unexpected behavior when using this option, the **Debugger>Setup** option **Run to** should be deselected.

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

Suppress download

Disables the downloading of code, while preserving the present content of the flash. This command is useful if you want to debug an application that already resides in target memory.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Use flash loader(s)

Use this option to use one or several flash loaders for downloading your application to flash memory. If a flash loader is available for the selected chip, it is used by default. Press the **Edit** button to display the **Flash Loader Overview** dialog box.

For more information about flash loaders, see *Flash loaders*, page 489.

Override default .board file

A default flash loader is selected based on your choice of device on the **General Optios>Target** page. To override the default flash loader, select **Override default .board file** and specify the path to the flash loader you want to use. A browse button is available for your convenience. Click **Edit** to display the **Flash Loader Overview** dialog box. For more information, see *Flash Loader Overview dialog box*, page 491.

Images

The **Images** options control the use of additional debug files to be downloaded.

The screenshot shows a dialog box titled "Images" with a light green background. It contains three identical sections, each for downloading an extra image. Each section starts with a checkbox labeled "Download extra image". Below each checkbox are three controls: a "Path:" text box with a browse button (...), an "Offset:" text box, and a "Debug info only" checkbox.

Download extra Images

Controls the use of additional debug files to be downloaded:

Path

Specify the debug file to be downloaded. A browse button is available for your convenience.

Offset

Specify an integer that determines the destination address for the downloaded debug file.

Debug info only

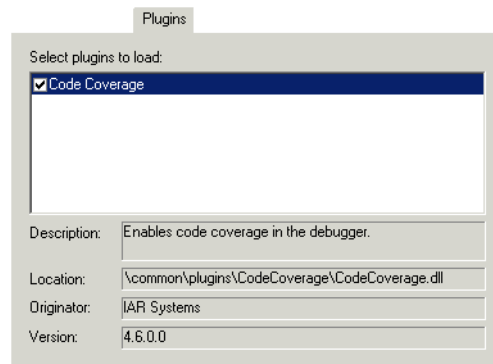
Makes the debugger download only debug information, and not the complete debug file.

If you want to download more than three images, use the related C-SPY macro, see `__loadImage`, page 403.

For more information, see *Loading multiple images*, page 53.

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 `arm\plugins` directory.

Originator

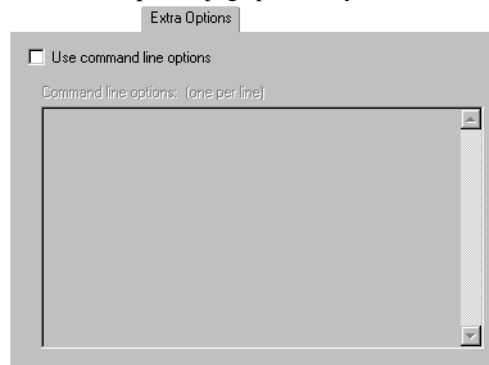
Informs about the originator of the plugin module, which can be modules provided by IAR Systems or by third-party vendors.

Version

Informs about the version number.

Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.

**Use command line options**

Specify command line arguments that are not supported by the IDE to be passed to C-SPY.

Note that it is possible to use the `/args` option to pass command line arguments to the debugged application.

Syntax: `/args arg0 arg1 ...`

Multiple lines with `/args` are allowed, for example:

```
/args --logfile log.txt
```

```
/args --verbose
```

If you use `/args`, these variables must be defined in your application:

```
/* __argc, the number of arguments in __argv. */
__no_init int __argc;

/* __argv, an array of pointers to strings that holds the
arguments; must be large enough to fit the number of
parameters.*/
__no_init const char * __argv[MAX_ARGS];

/* __argvbuf, a storage area for __argv; must be large enough to
hold all command line parameters. */
__no_init __root char __argvbuf[MAX_ARG_SIZE];
```

Multicore

The **Multicore** options configure multicore debugging.

Number of cores

For symmetric multicore debugging, specify the number of cores on your device.

Enable multicore master mode

Makes the debug session an asymmetric multicore debugger master. When you start a debug session, a new instance of the IAR Embedded Workbench IDE will be started, using the following options:

Port

Specify the TCP port (typically, larger than 1023) used for communication between the IDE instances.

Slave workspace

Specify the workspace to be opened in the slave instance.

Slave project

Specify the name of the project in the workspace to be opened in the slave instance. For example, if the project filename is `MySlaveProj.ewp`, specify `MySlaveProj`.

Slave configuration

Specify the build configuration to be used when debugging the slave. For example, `Debug` or `Release`.

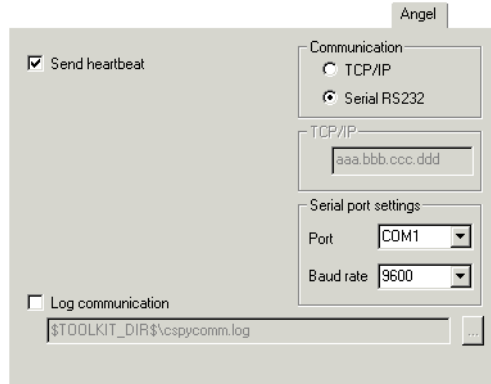
Reference information on C-SPY hardware debugger driver options

Reference information about:

- *Angel*, page 506
- *Setup options for CMSIS-DAP*, page 507
- *JTAG/SWD options for CMSIS-DAP*, page 510
- *GDB Server*, page 512
- *IAR ROM-monitor*, page 513
- *Setup options for I-jet/JTAGjet*, page 514
- *JTAG/SWD options for I-jet/JTAGjet*, page 517
- *Trace options for I-jet/JTAGjet*, page 519
- *Setup options for J-Link/J-Trace*, page 523
- *Connection options for J-Link/J-Trace*, page 528
- *Macraigor*, page 530
- *RDI*, page 532
- *Setup options for ST-LINK*, page 533
- *Setup options for TI Stellaris*, page 536
- *Setup options for TI XDS*, page 537
- *Third-Party Driver options*, page 539

Angel

The **Angel** options control the C-SPY Angel debug monitor driver.



Send heartbeat

Makes C-SPY poll the target system periodically while your application is running. That way, the debugger can detect if the target application is still running or has terminated abnormally. Enabling the heartbeat will consume some extra CPU cycles from the running program.

Communication

Selects the Angel communication link. RS232 serial port connection and TCP/IP via an Ethernet connection are supported.

TCP/IP

Specify the IP address of the target device in the text box.

Serial port settings

Configures the serial port. You can specify

- | | |
|------------------|---|
| Port | Selects which port on the host computer to use as the Angel communication link. |
| Baud rate | Sets the communication speed. |

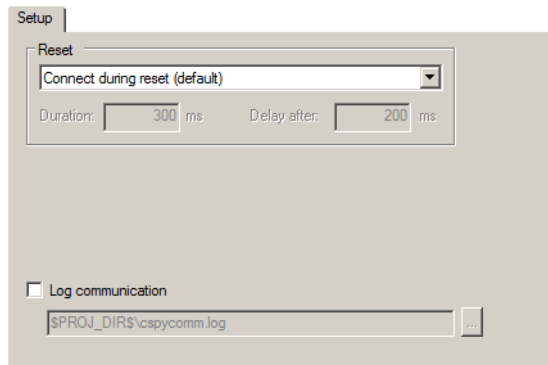
The initial Angel serial speed is always 9600 baud. After the initial handshake, the link speed is changed to the specified speed. Communication problems can occur at very high speeds; some Angel-based evaluation boards will not work above 38,400 baud.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the Angel monitor protocol is required.

Setup options for CMSIS-DAP

The **Setup** options control the C-SPY CMSIS-DAP driver.



Reset

Selects the reset strategy to be used when the debugger starts. Note that the **Reset** option is applicable only for Cortex-M devices. Based on your hardware, one of the strategies is the default. Choose between:

Disabled (no reset)	No reset is performed.
Software	Sets PC to the program entry address. This is a software reset.
Hardware	The probe toggles the <code>nSRST/nRESET</code> line on the JTAG connector to reset the device. This reset usually also resets the peripheral units. The reset pulse timing is controlled by the Duration and Delay after options. The processor should stop at the reset handler before executing any instruction. Some processors might not stop at the reset vector, but will be halted soon after, executing some instructions.
Core	Resets the core via the <code>VECTRESET</code> bit; the peripheral units are not affected.

System	Resets the core and peripherals.
Connect during reset	CMSIS-DAP connects to the target while keeping Reset active. Reset is pulled low and remains low while connecting to the target.
Custom	<p>Device-specific hardware reset. Some devices might require a special reset procedure or timing to enable debugging, or to bring the processor to a halt before it has executed any instruction.</p> <p>A watchdog timer might be disabled.</p> <p>Special debug modes, such as debugging in power-saving modes, might be turned on.</p> <p>This option is only available for some devices.</p>
Reset by watchdog or reset register	<p>Resets the processor using a software reset register or a watchdog reset. Peripheral units might not be reset.</p> <p>This reset strategy is recommended when the processor cannot be stopped at the reset vector using the hardware reset.</p> <p>Device-specific software reset. This option is only available for some devices.</p>
Reset and halt after bootloader	<p>Some devices have a ROM bootloader that executes before the processor jumps to your application code. Use this reset strategy to let the bootloader code execute and to halt the processor at the entry of the application code.</p> <p>Depending on the device, this reset strategy is implemented using the hardware, core, or system reset.</p> <p>This option is only available for some devices.</p>
Reset and halt before bootloader	<p>This reset strategy is complementary to the Reset and halt after bootloader strategy. Depending on the device, it is implemented using the hardware, core, or system reset.</p> <p>This option is only available for some devices.</p>

All of these strategies are available for both the JTAG and the SWD interface, and all strategies halt the CPU after the reset.

A software reset of the target does not change the settings of the target system; it only resets the program counter.

Normally, a C-SPY reset is a software reset only. If you use the **Hardware** option, C-SPY will generate an initial hardware reset when the debugger is started. This is performed once before download, and if the option **Use flash loader(s)** is selected, also once after flash download, see *Debugging code in flash*, page 58, and *Debugging code in RAM*, page 59.



Hardware resets can be a problem if the low-level setup of your application is not complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 56.

Duration

The time in milliseconds that the hardware reset asserts the reset signal (line `nSRST/nRESET`) low to reset the device.

Some devices might require a longer reset signal than the default 200 ms.

This option applies to the hardware reset, and to those custom reset strategies that use the hardware reset.

Delay after

The delay time, in milliseconds, after the reset signal has been de-asserted, before the debugger attempts to control the processor.

The processor might be kept internally in reset for some time after the external reset signal has been de-asserted, thus inaccessible for the debugger.

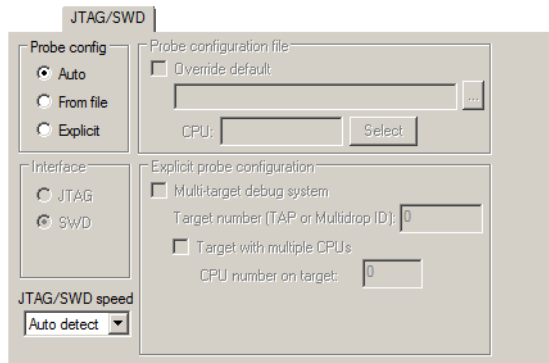
This option applies to the hardware reset, and to those custom reset styles that use the hardware reset.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

JTAG/SWD options for CMSIS-DAP

The JTAG/SWD options specify the interface between CMSIS-DAP and the target system.



Probe config

Auto

The CMSIS-DAP driver automatically identifies the target CPU. It uses the default probe configuration file, if there is one.

This works best if there is only one CPU present.

From file

Specifies that the probe configuration file needs to be overridden, or that there are several target CPUs.

Explicit

Specify how to find the target CPU.

Interface

Selects the communication interface between the debug probe and the target system. Choose between:

JTAG Uses the JTAG interface.

SWD Uses the SWD interface.

JTAG/SWD speed

Specify the JTAG and SWD communication speed. Choose between:

Auto detect Automatically uses the highest possible frequency for reliable operation.

Adaptive	Synchronizes the clock to the processor clock outside the core. Works only with ARM devices that have the <code>RTCK</code> JTAG signal available.
<i>n</i> MHz	<p>Sets the JTAG and SWD communication speed to the selected frequency.</p> <p>If there are JTAG communication problems or problems in writing to target memory (for example during program download), these problems might be resolved if the speed is set to a lower frequency.</p>

Probe configuration file

Override default

Specify a probe configuration file to be used instead of the default probe configuration file that comes with the product package.

Select

Specify how to find the target CPU.

Explicit probe configuration

Multi-target debug system

Specifies that the debug system consists of more than one CPU.

Target number (TAP or Multidrop ID)

If the debug system is a multi-drop SWD, specify the Multidrop ID (in hexadecimal notation) of the DAP where your CPU is located.

If the debug system is a JTAG scan chain, specify the **Target number TAP** (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero. If there are several CPUs at the TAP position, you also need to specify the **CPU number on target**.

CPU number on target

If the debug system is a multi-core SWD, specify the CPU number on the DAP.

GDB Server

The **GDB Server** options control the C-SPY GDB Server for the STR9-comStick evaluation board.



TCP/IP address or hostname

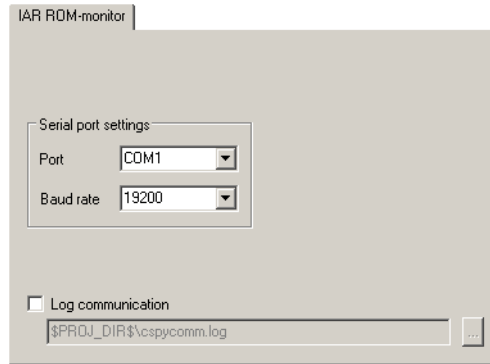
Specify the IP address and port number of a GDB server; by default the port number 3333 is used. The TCP/IP connection is used for connecting to a J-Link server running on a remote computer.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the JTAG interface is required.

IAR ROM-monitor

The **IAR ROM-monitor** options control the C-SPY IAR ROM-monitor interface.



Serial port settings

Configures the serial port. You can specify

- | | |
|------------------|---|
| Port | Selects which port on the host computer to use as the ROM-monitor communication link. |
| Baud rate | Sets the communication speed. The serial port communication link speed must match the speed selected on the target board. |

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, a detailed knowledge of the ROM-monitor protocol is required.

Setup options for I-jet/JTAGjet

The **Setup** options control the I-jet and I-jet Trace in-circuit debugging probes and the JTAGjet debug probe.

The screenshot shows a 'Setup' dialog box. Under the 'Reset' section, there is a dropdown menu currently showing 'Disabled (no reset)'. Below it are two input fields: 'Duration:' with the value '300' and 'ms', and 'Delay after:' with the value '200' and 'ms'. The 'Target power' section contains three radio buttons: 'From the probe' (which is checked), 'Leave on after debugging', and 'Switch off after debugging'. At the bottom, there is a checkbox for 'Log communication' and a text field containing the path '\$PROJ_DIRS\cspycomm.log' with a browse button ('...').

Reset

Selects the reset strategy to be used when the debugger starts. Note that Cortex-M uses a different set of strategies than other devices. Based on your hardware, one of the strategies is the default. Choose between:

Disabled (no reset)	No reset is performed.
Software	Sets <code>PC</code> to the program entry address and <code>SP</code> to the initial stack pointer value. This is a software reset.
Hardware	The probe toggles the <code>nSRST/nRESET</code> line on the JTAG connector to reset the device. This reset usually also resets the peripheral units. The reset pulse timing is controlled by the Duration and Delay after options. The processor should stop at the reset handler before executing any instruction. Some processors might not stop at the reset vector, but will be halted soon after, executing some instructions.
Core	Resets the core via the <code>VECTRESET</code> bit; the peripheral units are not affected. For Cortex-M devices only.
System	Resets the core and peripheral units by setting the <code>SYSRESETREQ</code> bit in the <code>AIRCR</code> register. Reset vector catch is used for stopping the CPU at the reset vector before the first instruction is executed. For Cortex-M devices only.

Connect during reset	I-jet/JTAGjet connects to the target while keeping Reset active. Reset is pulled low and remains low while connecting to the target. This is the recommended reset strategy for STM32 devices.
Custom	<p>Device-specific hardware reset. Some devices might require a special reset procedure or timing to enable debugging, or to bring the processor to a halt before it has executed any instruction.</p> <p>A watchdog timer might be disabled.</p> <p>Special debug modes, such as debugging in power-saving modes, might be turned on.</p> <p>This option is only available for some devices.</p>
Reset by watchdog or reset register	<p>Resets the processor using a software reset register or a watchdog reset. Peripheral units might not be reset.</p> <p>This reset strategy is recommended when the processor cannot be stopped at the reset vector using the hardware reset.</p> <p>Device-specific software reset. This option is only available for some devices.</p>
Reset and halt after bootloader	<p>Some devices have a ROM bootloader that executes before the processor jumps to your application code. Use this reset strategy to let the bootloader code execute and to halt the processor at the entry of the application code.</p> <p>Depending on the device, this reset strategy is implemented using the hardware, core, or system reset.</p> <p>This option is only available for some devices.</p>
Reset and halt before bootloader	<p>This reset strategy is complementary to the Reset and halt after bootloader strategy. Depending on the device, it is implemented using the hardware, core, or system reset.</p> <p>This option is only available for some devices.</p>

All of these strategies are available for both the JTAG and the SWD interface, and all strategies halt the CPU after the reset.

A software reset of the target does not change the settings of the target system; it only resets the program counter and the mode register CPSR to its reset state. For some ARM9, ARM11, and Cortex-A devices, it also resets the CP15 system control

coprocessor, effectively disabling the virtual memory (MMU), caches and memory protection.

Normally, a C-SPY reset is a software reset only. If you use the **Hardware** option, C-SPY will generate an initial hardware reset when the debugger is started. This is performed once before download, and if the option **Use flash loader(s)** is selected, also once after flash download, see *Debugging code in flash*, page 58, and *Debugging code in RAM*, page 59.



Hardware resets can be a problem if the low-level setup of your application is not complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 56.

Duration

The time in milliseconds that the hardware reset asserts the reset signal (line `nSRST/nRESET`) low to reset the device.

Some devices might require a longer reset signal than the default 200 ms.

This option applies to the hardware reset, and to those custom reset strategies that use the hardware reset.

Delay after

The delay time, in milliseconds, after the reset signal has been de-asserted, before the debugger attempts to control the processor.

The processor might be kept internally in reset for some time after the external reset signal has been de-asserted, thus inaccessible for the debugger.

This option applies to the hardware reset, and to those custom reset styles that use the hardware reset.

Target power

If power for the target system is supplied from the probe, this option specifies the status of the power supply after debugging. Choose between:

Leave on after debugging	Continues to supply power to the target even after the debug session has been stopped.
Switch off after debugging	Turns off the power to the target when the debug session stops.

Log communication

Logs trace output, that is a sequence of internal communication activities, between C-SPY and its lower-level interfaces to hardware. This log is primarily useful as a troubleshooting help when contacting IAR Systems support.

JTAG/SWD options for I-jet/JTAGjet

The JTAG/SWD options specify the interface between I-jet, I-jet Trace, or JTAGjet and the target system.

Probe config

Auto

The I-jet/JTAGjet driver automatically identifies the target CPU. It uses the default probe configuration file, if there is one.

This works best if there is only one CPU present.

From file

Specifies that the probe configuration file needs to be overridden, or that there are several target CPUs.

Explicit

Specify how to find the target CPU.

Interface

Selects the communication interface between the debug probe and the target system. Choose between:

JTAG

Uses the JTAG interface.

SWD Uses the SWO interface, which uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select **stdout/stderr via SWO** on the **General Options>Library Configuration** page, SWD is selected automatically. For more information about SWO settings, see *SWO Trace Window Settings dialog box*, page 212.

JTAG/SWD speed

Specify the JTAG and SWD communication speed. Choose between:

Auto detect Automatically uses the highest possible frequency for reliable operation.

Adaptive Synchronizes the clock to the processor clock outside the core. Works only with ARM devices that have the `RTCK` JTAG signal available.

***n* MHz** Sets the JTAG and SWD communication speed to the selected frequency.

If there are JTAG communication problems or problems in writing to target memory (for example during program download), these problems might be resolved if the speed is set to a lower frequency.

Probe configuration file

Override default

Specify a probe configuration file to be used instead of the default probe configuration file that comes with the product package.

Select

Specify how to find the target CPU.

Explicit probe configuration

Multi-target debug system

Specifies that the debug system consists of more than one CPU.

Target number (TAP or Multidrop ID)

If the debug system is a multi-drop SWD, specify the Multidrop ID (in hexadecimal notation) of the DAP where your CPU is located.

If the debug system is a JTAG scan chain, specify the **Target number TAP** (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero. If there are several CPUs at the TAP position, you also need to specify the **CPU number on target**.

CPU number on target

If the debug system is a multi-core SWD, specify the CPU number on the DAP.

JTAG scan chain contains non-ARM devices

Enables JTAG scan chains that mix ARM devices with other devices like, for example, FPGA.

Preceding bits

Specify the TAP (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero.

Trace options for I-jet/JTAGjet

The Trace options specify the trace behavior for I-jet/JTAGjet.

The screenshot shows a 'Trace' dialog box with the following settings:

- Trace data collection:**
 - Mode: Auto
 - Allow ETB:
 - Buffer limit: 8 Msamples
- SWO protocol:**
 - Auto (selected)
 - Manchester
 - UART
- SWO clock setup:**
 - CPU clock: [] MHz
 - SWD prescaler: Auto
- SWO on the TraceD0 pin

Mode

Power measurement (either TrgPwr as provided by the probe or via I-scope) does not depend on a particular trace mode and is always possible (if the probe supports it).

The **Debug Log** window will include messages about the currently used trace mode. If a particular mode cannot be used, either due to probe or board/device limitations, trace will be disabled and a warning message will be displayed in the **Debug Log** window. This is how the support of a particular trace mode is checked:

- The probe must support the particular mode.
- The probe must support the particular mode on a specific core. For example, ETM on ARM9 is not supported by the I-jet Trace probe.

- The specific core must support the particular mode. For example, Cortex-M0 does not support SWO/ETM/ETB at all and ARM9 does not support SWO.
- The used adapter must support the specified mode. For example, ETM trace is not possible when the ARM20 adapter is used with I-jet Trace.
- The specific device must support the particular mode. For example, ETM trace is not possible on a Cortex-M3 without ETM, which cannot be detected until reading the on-chip TPIU configuration register.

The **Mode** option specifies the mechanism and interface for trace data collection. Choose between:

Auto

Automatically selects the best possible mechanism and interface, depending on probe and board/device capabilities.

The basic modes are tried in probe-dependent order:

- I-jet: First SWO, then ETB (ETM is not supported).
- I-jet Trace: First ETM, then SWO, then ETB.
- JTAGjet-Trace: First ETM, then ETB (SWO is not supported).
- JTAGjet: Only ETB (SWO and ETM are not supported).

If none of these modes are available, trace will be disabled (as when **None** is selected). In **Auto** mode, more initial accesses to trace-related on-chip resources might be made. So, if you are using a specific probe and a specific mode, you might want to set the mode explicitly which will make C-SPY initialize/configure trace resources more efficiently.

None

Disables trace. In this mode, C-SPY will not access any trace-related on-chip resources. You can use this mode when:

- You are experiencing connectivity problems. It might be easier to diagnose the reason for connectivity problems without the interference from initialization of trace resources.
- Trace might change some internal clocking and/or GPIO mux settings and as a result some applications might not work well with a specific trace mode.
- You want to exercise low-power modes. Internal on-chip trace logic and toggling trace pins will require some additional current and it might interfere with low-power measurements. In extreme cases, enabling clocks for trace/GPIO might prevent the CPU from actually entering low-power modes, because some clocks inside the CPU must be kept active.

Serial (SWO)

Collects trace data through the serial (SWO) interface.

Parallel (ETM)

Collects trace data through the parallel (ETM) interface.

On-chip (ETB/MTB)

Collects trace data through the on-chip (ETB/MTB) interface.

Allow ETB

Allows simultaneous on-chip (ETB) trace. This option is only available when **Mode** is **Serial (SWO)**.

Buffer limit

Select the number of mega-samples to collect. This option is only available if parallel (ETM) mode is used, either explicitly through **Parallel (ETM)** or implicitly through **Auto**.

ETM provides 4-bit trace data on each edge of the trace clock. Such a smallest amount of trace data is called an *ETM sample*. The trace probe can collect 4-bit trace from ETM-capable devices over a MIPI20 connector. Collected trace data is stored inside the probe.

Because reading and decoding large amounts of trace data might take some time, it is possible to limit what portion of ETM memory will actually be read by C-SPY once trace data collection is stopped (either because the CPU stopped or because the buffer got full). Use the **Buffer limit** option to set this limit to a certain number of Msamples. (1 Msample = ~1 million 4-bit samples) The higher **Buffer limit**, the more useful trace data will be available, but it might take longer to see results and more memory to store it. C-SPY will retrieve the most recent samples from the trace probe, and the rest of the collected trace data will be discarded by the probe.

There is no simple correlation between the number of raw ETM samples and the number of PC samples visible in the ETM Trace window. The ETM protocol itself is highly compressed, and the probe provides additional compression of ETM idle cycles, so it is not possible to guess how many instructions can be decoded from a certain number of raw ETM samples collected by the trace probe. If your application changes PC a lot, ETM will need to use more samples to send more PC bits and as such, trace data will not compress well. For a particular application profile, this number is usually constant (between 0.5 and 2 instructions for each 4-bit sample), so you must use your own judgment to see what buffer limit that provides good balance between the size of decoded data and C-SPY performance.

Note: For the JTAGjet-Trace probe, this option is not available. The buffer limit for JTAGjet-Trace is fixed to 1M/2M/4M samples, depending on hardware limitations.

SWO protocol

Specifies the communication protocol for the SWO channel. Choose between:

Auto	Automatically selects the best possible protocol and speed, depending on the device you are using.
Manchester	Specifies the Manchester protocol.
UART	Specifies the UART protocol.

CPU clock

Specifies the exact clock frequency used by the internal processor clock, `HCLK`, in MHz. The value can have decimals. This value is used for configuring the SWO communication speed.

SWO prescaler

Specifies the clock prescaler of the SWO communication channel in KHz. The prescaler, in turn, determines the SWO clock frequency.

Auto automatically uses the highest possible frequency that the I-jet or I-jet Trace debug probe can handle. Use this setting if data packets are lost during transmission.

To override the **SWO clock setup** options, use the **Override project default** option in the **SWO Configuration** dialog box, see *Override project default*, page 216.

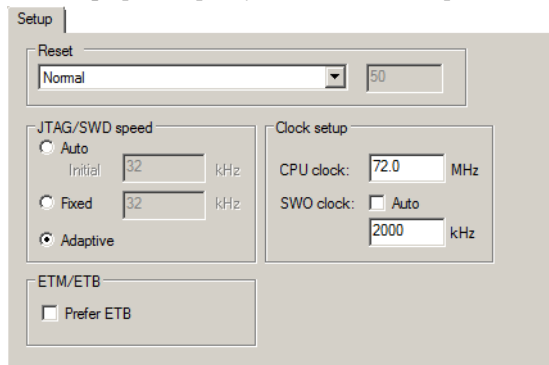
SWO on the TraceD0 pin

Specifies that SWO trace data is output on the trace data D0 pin. When using this option, both the SWD and the JTAG interface can handle SWO trace data.

Note that both the device and the board you are using must support this pin.

Setup options for J-Link/J-Trace

The **Setup** options specify the J-Link/J-Trace probe.



Reset

Selects the reset strategy to be used when the debugger starts. Note that Cortex-M uses a different set of strategies than other devices. The actual reset strategy type number is specified for each available choice. Choose between:

- | | |
|---------------------------------|---|
| Normal (0, default) | This is the default strategy. It does whatever is the best way to reset the target device, which for most devices is the same as the reset strategy Core and peripherals (8). Some special handling might be needed for certain devices, for example devices which have a ROM bootloader that needs to run after reset and before your application is started. |
| Core (1) | Resets the core via the VECTRESET bit; the peripheral units are not affected. |
| Core and peripherals (8) | Resets the core and the peripherals. |
| Reset Pin (2) | J-Link pulls its RESET pin low to reset the core and the peripheral units. Normally, this causes the CPU RESET pin of the target device to go low as well, which results in a reset of both the CPU and the peripheral units. |
| Connect during reset (3) | J-Link connects to the target while keeping Reset active (reset is pulled low and remains low while connecting to the target). This is the recommended reset strategy for STM32 devices. This strategy is available for STM32 devices only. |

Halt after bootloader (4 or 7) NXP Cortex-M0 devices. This is the same strategy as the Normal strategy, but the target is halted when the bootloader has finished executing. This is the recommended reset strategy for LPC11xx and LPC13xx devices.

Analog Devices Cortex-M3 devices (7), Resets the core and peripheral units by setting the `SYSRESETREQ` bit in the AIRCR. The core is allowed to perform the ADI kernel (which enables the debug interface), but the core is halted before the first instruction after the kernel is executed to guarantee that no user application code is performed after reset.

Halt before bootloader (5) This is the same strategy as the Normal strategy, but the target is halted before the bootloader has started executing. This strategy is normally not used, except in situations where the bootloader needs to be debugged. This strategy is available for LPC11xx and LPC13xx devices only.

Normal, disable watchdog (6, 9, or 10) First performs a Normal reset, to reset the core and peripheral units and halt the CPU immediately after reset. After the CPU is halted, the watchdog is disabled, because the watchdog is by default running after reset. If the target application does not feed the watchdog, J-Link loses connection to the device because it is permanently reset. This strategy is available for Freescale Kinetis devices (6), for NXP LPC 1200 devices (9), and for Samsung S3FN60D devices (10).

All of these strategies are available for both the JTAG and the SWD interface, and all strategies halt the CPU after the reset.

For other cores, choose between these strategies:

Hardware, halt after delay (ms) (0) Specify the delay between the hardware reset and the halt of the processor. This is used for making sure that the chip is in a fully operational state when C-SPY starts to access it. By default, the delay is set to zero to halt the processor as quickly as possible.

This is a hardware reset.

Hardware, halt using Breakpoint (1)	<p>After reset, J-Link continuously tries to halt the CPU using a breakpoint. Typically, this halts the CPU shortly after reset; the CPU can in most systems execute some instructions before it is halted.</p> <p>This is a hardware reset.</p>
Hardware, halt at 0 (4)	<p>Halts the processor by placing a breakpoint at the address zero. Note that this is not supported by all ARM microcontrollers.</p> <p>This is a hardware reset.</p>
Hardware, halt using DBGRQ (5)	<p>After reset, J-Link continuously tries to halt the CPU using DBGRQ. Typically, this halts the CPU shortly after reset; the CPU can in most systems execute some instructions before it is halted.</p> <p>This is a hardware reset.</p>
Software (-)	<p>Sets PC to the program entry address.</p> <p>This is a software reset.</p>
Software, Analog devices (2)	<p>Uses a reset sequence specific for the Analog Devices ADuC7xxx family. This strategy is only available if you have selected such a device from the Device drop-down list on the General Options>Target page.</p> <p>This is a software reset.</p>
Hardware, NXP LPC (9)	<p>This strategy is only available if you have selected such a device from the Device drop-down list on the General Options>Target page.</p> <p>This is a hardware reset specific to NXP LPC devices.</p>
Hardware, Atmel AT91SAM7 (8)	<p>This strategy is only available if you have selected such a device from the Device drop-down list on the General Options>Target page.</p> <p>This is a hardware reset specific for the Atmel AT91SAM7 family.</p>

For more details about the different reset strategies, see the *IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores* available in the `arm\doc` directory.

A software reset of the target does not change the settings of the target system; it only resets the program counter and the mode register CPSR to its reset state. Normally, a

C-SPY reset is a software reset only. If you use the Hardware reset option, C-SPY will generate an initial hardware reset when the debugger is started. This is performed once before download, and if the option Use flash loader(s) is selected, also once after flash download, see *Debugging code in flash*, page 58, and *Debugging code in RAM*, page 59.



Hardware resets can be a problem if the low-level setup of your application is not complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 56.

JTAG/SWD speed

Specify the JTAG communication speed in kHz. Choose between:

Auto	<p>Automatically uses the highest possible frequency for reliable operation. The initial speed is the fixed frequency used until the highest possible frequency is found. The default initial frequency—32 kHz—can normally be used, but in cases where it is necessary to halt the CPU after the initial reset, in as short time as possible, the initial frequency should be increased.</p> <p>A high initial speed is necessary, for example, when the CPU starts to execute unwanted instructions—for example power down instructions—from flash or RAM after a reset. A high initial speed would in such cases ensure that the debugger can quickly halt the CPU after the reset.</p> <p>The initial value must be in the range 1–12000 kHz.</p>
Fixed	<p>Sets the JTAG communication speed in kHz. The value must be in the range 1–12000 kHz.</p> <p>If there are JTAG communication problems or problems in writing to target memory (for example during program download), these problems might be resolved if the speed is set to a lower frequency.</p>
Adaptive	<p>Synchronizes the clock to the processor clock outside the core. Works only with ARM devices that have the <code>RTCK</code> JTAG signal available. For more information about adaptive speed, see the <i>IAR J-Link and IAR J-Trace User Guide for JTAG Emulators for ARM Cores</i> available in the <code>arm\doc</code> directory.</p>

Clock setup

Specifies the CPU clock. Choose between:

- | | |
|------------------|--|
| CPU clock | Specifies the exact clock frequency used by the internal processor clock, <code>HCLK</code> , in MHz. The value can have decimals. This value is used for configuring the SWO communication speed and for calculating timestamps. |
| SWO clock | Specifies the clock frequency of the SWO communication channel in KHz. |
| Auto | Automatically uses the highest possible frequency that the debug probe can handle. If Auto is not selected, the wanted SWO clock value can be input in the text box. The value can have decimals. Use this option if data packets are lost during transmission. |

To override the **Clock setup** options, use the **Override project default** option in the **SWO Configuration** dialog box, see *SWO Configuration dialog box*, page 214.

ETM/ETB

The **Prefer ETB** option selects ETB trace instead of ETM trace, which is the default.

Note: This option applies only to J-Trace.

Connection options for J-Link/J-Trace

The **Connection** options specify the connection with the J-Link/J-Trace probe.

Communication

Selects the communication channel between C-SPY and the J-Link debug probe. Choose between:

- USB** Selects the USB connection. If Serial number is selected in the drop-down list, the J-Link debug probe with the specified serial number is chosen.
- TCP/IP** Specify the IP address of a J-Link server. The TCP/IP connection is used for connecting to a J-Link server running on a remote computer.
- IP address**, specify the IP address of a J-Link probe connected to LAN.
- Auto detect**, automatically scans the network for J-Link probes. Use the dialog box to choose among the detected J-Link probes.
- Serial number**, connects to the J-Link probe on the network with the serial number that you specify.

Interface

Selects the communication interface between the J-Link debug probe and the target system. Choose between:

- JTAG (default)** Uses the JTAG interface.

SWD Uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the **General Options>Library Configuration** page, SWD is selected automatically. For more information about SWO settings, see *SWO Trace Window Settings dialog box*, page 212.

JTAG scan chain

Specifies the JTAG scan chain. Choose between:

JTAG scan chain with multiple targets	Specifies that there is more than one device on the JTAG scan chain.
TAP number	Specify the TAP (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero.
Scan chain contains non-ARM devices	Enables JTAG scan chains that mix ARM devices with other devices like, for example, FPGA.
Preceding bits	Specify the number of IR bits before the ARM device to be debugged.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

Macraigor

The **Macraigor** options specify the Macraigor interface.

OCD interface device

Selects the device corresponding to the hardware interface you are using. Supported Macraigor JTAG probes is Macraigor **mpDemon**.

Interface

Selects the communication interface between the J-Link debug probe and the target system. Choose between:

JTAG (default)

Uses the JTAG interface.

SWD

Uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the **General Options>Library Configuration** page, SWD is selected automatically. For more information about SWO settings, see *SWO Trace Window Settings dialog box*, page 212.

JTAG speed

Specify the speed between the JTAG probe and the ARM JTAG ICE port. The number must be in the range 1–8 and sets the factor by which the JTAG probe clock is divided when generating the scan clock.



The mpDemon interface might require a higher setting such as 2 or 3, that is, a lower speed.

TCP/IP

Specify the IP address of a JTAG probe connected to the Ethernet/LAN port.

Port

Selects which serial port or parallel port on the host computer to use as communication link. Select the host port to which the JTAG probe is connected.

In the case of parallel ports, you should normally use LPT1 if the computer is equipped with a single parallel port. Note that a laptop computer might in some cases map its single parallel port to LPT2 or LPT3. If possible, configure the parallel port in EPP mode because this mode is fastest; bidirectional and compatible modes will work but are slower.

Baud rate

Selects the serial communication speed.

Hardware reset

Generates an initial hardware reset when the debugger is started. This is performed once before download, and if the option **Use flash loader(s)** is selected, also once after flash download, see *Debugging code in flash*, page 58, and *Debugging code in RAM*, page 59.

A software reset of the target does not change the settings of the target system; it only resets the program counter to its reset state. Normally, a C-SPY reset is a software reset only.



Hardware resets can be a problem if the low-level setup of your application is not complete. If low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 56.

JTAG scan chain with multiple targets

Defines each device on the JTAG scan chain, if there is more than one. Also, you must state which device you want to connect to. The syntax is:

```
<0>@dev0, dev1, dev2, dev3, . . .
```

where 0 is the TAP number of the device to connect to, and `dev0` is the nearest TDO pin on the Macraigor JTAG probe.

Debug handler address

Specify the location—the memory address—of the debug handler used by Intel XScale devices. To save memory space, you should specify an address where a small portion of cache RAM can be mapped, which means the location should not contain any physical memory. Preferably, find an unused area in the lower 16-Mbyte memory and place the handler address there.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

RDI

With the **RDI** options you can use JTAG interfaces compliant with the ARM Ltd. RDI 1.5.1 specification. One example of such an interface is the ARM RealView Multi-ICE JTAG interface.

Manufacturer RDI driver

Specify the file path to the RDI driver DLL file provided with the JTAG pod.

Allow hardware reset

Allows the emulator to perform a hardware reset of the target.

A software reset of the target does not change the settings of the target system; it only resets the program counter to its reset state.



You should only allow hardware resets if the low-level setup of your application is complete. If the low-level setup does not set up memory configuration and clocks, the application will not work after a hardware reset. To handle this in C-SPY, the setup macro function `execUserReset()` is suitable. For a similar example where `execUserPreload()` is used, see *Remapping memory*, page 56.

Note: This option requires that hardware resets are supported by the RDI driver you are using.

Catch exceptions

Causes exceptions to be treated as breakpoints. Instead of handling the exception as defined by the running program, the debugger will stop.

The ARM core exceptions that can be caught are:

Exception	Description
Reset	Reset
Undef	Undefined instruction
SWI	Software interrupt
Data	Data abort (data access memory fault)
Prefetch	Prefetch abort (instruction fetch memory fault)
IRQ	Normal interrupt
FIQ	Fast interrupt

Table 53: Catching exceptions

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

Setup options for ST-LINK

The **Setup** options specify the ST-LINK interface.

Emulator

Specify the emulator you are using. To force the **Debug Probe Selection** dialog box to be displayed each time you start a debug session, use the option **Always prompt for probe selection**.

Reset

Selects the reset strategy to be used when the debugger starts. The actual reset strategy type number is specified for each available choice. Choose between:

- | | |
|---------------------------------|---|
| Normal (0) | Performs the standard reset procedure. |
| Reset Pin (1) | Uses the reset pin to perform a hardware reset. Only available for ST-LINK version 2. |
| Connect during reset (2) | ST-LINK connects to the target while keeping the reset pin active (the reset pin is pulled low and remains low while connecting to the target). Only available for ST-LINK version 2. |

Interface

Selects the communication interface between the ST-LINK debug probe and the target system. Choose between:

- | | |
|-----------------------|----------------------------|
| JTAG (default) | Uses the JTAG interface. |
| SWD | Uses fewer pins than JTAG. |

JTAG/SWD speed

Specify the JTAG and SWD communication speed. Choose between:

- | | |
|---------------------|--|
| Auto detect | Automatically uses the highest possible frequency for reliable operation. |
| Adaptive | Synchronizes the clock to the processor clock outside the core. Works only with ARM devices that have the <code>RTCK</code> JTAG signal available. |
| <i>n</i> MHz | Sets the JTAG and SWD communication speed to the selected frequency. |

If there are JTAG communication problems or problems in writing to target memory (for example during program download), these problems might be resolved if the speed is set to a lower frequency.

Communication options for ST-LINK

The **Communication** options specify the ST-LINK interface.

The screenshot shows a dialog box titled "Communication". Inside, there is a "Clock setup" section with the following controls:

- "CPU clock:" followed by a text input field and "MHz".
- "SWO clock:" followed by a radio button labeled "Auto" and a text input field containing "2000" followed by "kHz".

Below the "Clock setup" section, there is a checkbox labeled "Log communication" and a text input field containing the path "\$PROJ_DIR\$\cspycomm.log" with a browse button (three dots) to its right.

Clock setup

Specifies the CPU clock. Choose between:

- | | |
|------------------|--|
| CPU clock | Specifies the exact clock frequency used by the internal processor clock, <code>HCLK</code> , in MHz. The value can have decimals. This value is used for configuring the SWO communication speed and for calculating timestamps. |
| SWO clock | Specifies the clock frequency of the SWO communication channel in KHz. |
| Auto | Automatically uses the highest possible frequency that the debug probe can handle. If Auto is not selected, the wanted SWO clock value can be input in the text box. The value can have decimals. Use this option if data packets are lost during transmission. |

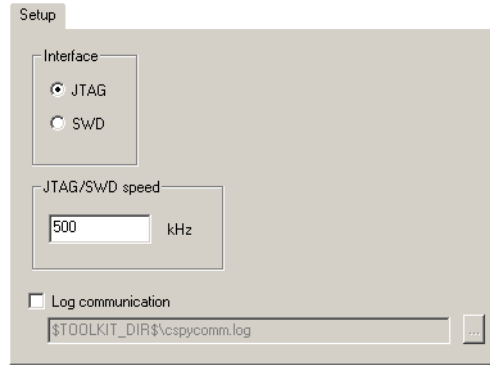
To override the **Clock setup** options, use the **Override project default** option in the **SWO Configuration** dialog box, see *Override project default*, page 216.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

Setup options for TI Stellaris

The **Setup** options specify the TI Stellaris interface.



Interface

Selects the communication interface between the J-Link debug probe and the target system. Choose between:

JTAG (default)

Uses the JTAG interface.

SWD

Uses fewer pins than JTAG. Select SWD if you want to use the serial-wire output (SWO) communication channel. Note that if you select stdout/stderr via SWO on the **General Options>Library Configuration** page, SWD is selected automatically. For more information about SWO settings, see *SWO Trace Window Settings dialog box*, page 212.

JTAG/SWD speed

Specify the JTAG communication speed in kHz.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

Setup options for TI XDS

The **Setup** options control the TI XDS interface.

Emulator

Specify the emulator you are using. If more than one debug probe is connected to the host computer, use **Serial no** and **Select probe** to make the proper selection. To override the default board file, specify a board file using the **Board file** option.

Reset

Select the reset strategy to be used when C-SPY starts.

Interface

Select the communication interface between the XDS debug probe and the target system.

JTAG/SWD speed

Specify the JTAG communication speed.

TI emulation package installation path

Select **Override default** to override the default installation path of the Texas Instruments emulation package.

Communication options for TI XDS

The **Communication** options control the TI XDS interface.

Clock setup

Specifies the CPU clock. Choose between:

CPU clock

Specifies the exact clock frequency used by the internal processor clock, `HCLK`, in MHz. The value can have decimals. This value is used for configuring the SWO communication speed and for calculating timestamps.

SWO clock

Specifies the clock frequency of the SWO communication channel in KHz.

Auto

Automatically uses the highest possible frequency that the J-Link debug probe can handle. If **Auto** is not selected, the wanted SWO clock value can be input in the text box. The value can have decimals. Use this option if data packets are lost during transmission.

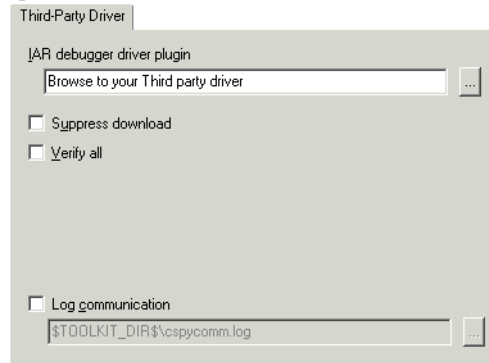
To override the **Clock setup** options, use the **Override project default** option in the **SWO Configuration** dialog box, see *SWO Configuration dialog box*, page 214.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

Third-Party Driver options

The **Third-Party Driver** options are used for loading any driver plugin provided by a third-party vendor. These drivers must be compatible with the C-SPY debugger driver specification.



In addition to the options you can set here, you can set options for the third-party driver using the **Project>Options>Debugger>Extra Options** page.

IAR debugger driver plugin

Specify the file path to the third-party driver plugin DLL file. A browse button is available for your convenience.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

Additional information on C-SPY drivers

This chapter describes the additional menus and features provided by the C-SPY® drivers. You will also find some useful hints about resolving problems.

Reference information on C-SPY driver menus

This section gives reference information on the menus specific to the C-SPY drivers. More specifically, this means:

- *C-SPY driver*, page 541
- *Simulator menu*, page 542
- *CMSIS-DAP menu*, page 544
- *GDB Server menu*, page 545
- *I-jet/JTAGjet menu*, page 546
- *J-Link menu*, page 549
- *Macraigor JTAG menu*, page 552
- *RDI menu*, page 552
- *ST-LINK menu*, page 553
- *TI Stellaris menu*, page 554
- *TI XDS menu*, page 554

C-SPY driver

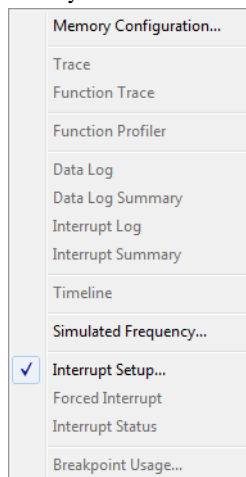
Before you start the C-SPY debugger, you must first specify a C-SPY driver in the **Options** dialog box, using the option **Debugger>Setup>Driver**.

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 Configuration

Displays a dialog box where you configure C-SPY to match the memory of your device, see *Memory Configuration dialog box, for the C-SPY simulator*, page 187.

Trace

Opens a window which displays the collected trace data, see *Trace window*, page 218.

Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 223.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 263.

Data Log

Opens a window which logs accesses to up to four different memory locations or areas, see *Data Log window*, page 117.

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 119.

Interrupt Log

Opens a window which displays the status of all defined interrupts, see *Interrupt Log window*, page 357.

Interrupt Log Summary

Opens a window which displays a summary of the status of all defined interrupts, see *Interrupt Log Summary window*, page 361.

Timeline

Opens a window which gives a graphical view of various kinds of information on a timeline, see *Timeline window*, page 224.

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.

Interrupt Setup

Displays a dialog box where you can configure C-SPY interrupt simulation, see *Interrupt Setup dialog box*, page 351.

Forced Interrupts

Opens a window from where you can instantly trigger an interrupt, see *Forced Interrupt window*, page 354.

Interrupt Status

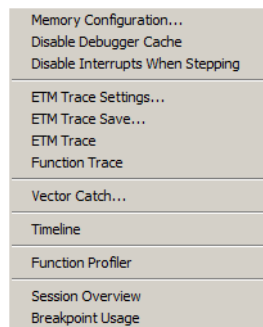
Opens a window from where you can instantly trigger an interrupt, see *Interrupt Status window*, page 355.

Breakpoint Usage

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 139.

CMSIS-DAP menu

When you are using the C-SPY CMSIS-DAP driver, the **CMSIS-DAP** menu is added to the menu bar.



Menu commands

These commands are available on the menu:

Memory Configuration

Displays a dialog box; see *Memory Configuration dialog box, in C-SPY hardware debugger drivers*, page 191.

Disable Debugger Cache

Disables memory caching and memory range checking in C-SPY.

Normally, C-SPY uses the memory range information in the **Memory Configuration** dialog box both to restrict access to certain parts of target memory and to cache target memory contents for improved C-SPY performance. Under certain rare circumstances, this is not appropriate, and you can choose **Disable Debugger Cache** to turn off the caching and memory range checking completely. All accesses from C-SPY will then result in corresponding accesses to the target system. Some of those circumstances are:

- When memory is remapped at runtime and cannot be specified as a fixed set of ranges.
- When the memory range setup is incorrect or incomplete.

Disable Interrupts When Stepping

Ensures that only the stepped instructions will be executed. Interrupts will not be executed. This command can be used when not running at full speed and some interrupts interfere with the debugging process.

ETM Trace Settings

Displays a dialog box; see *ETM Trace Settings dialog box*, page 208.

ETM Trace Save

Displays a dialog box; see *Trace Save dialog box*, page 253.

ETM Trace

Opens the ETM Trace window; see *Trace window*, page 218.

Function Trace

Opens a window; see *Function Trace window*, page 223.

Vector Catch

Displays a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see *Vector Catch dialog box*, page 156. Note that this command is not available for all ARM cores.

Timeline

Opens a window; see *Timeline window*, page 224.

This menu command is only available when the SWD/SWO interface is used.

Function Profiler

Opens a window which shows timing information for the functions; see *Function Profiler window*, page 263.

Session Overview

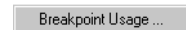
Displays a window that lists information about the debug session, such as details about project settings, session settings, and the session state. To save the contents of the window to a file, choose **Save As** from the context menu.

Breakpoint Usage

Opens a window which lists all active breakpoints; see *Breakpoint Usage window*, page 139.

GDB Server menu

When you are using the C-SPY GDB Server driver, the **GDB Server** menu is added to the menu bar.

**Menu commands**

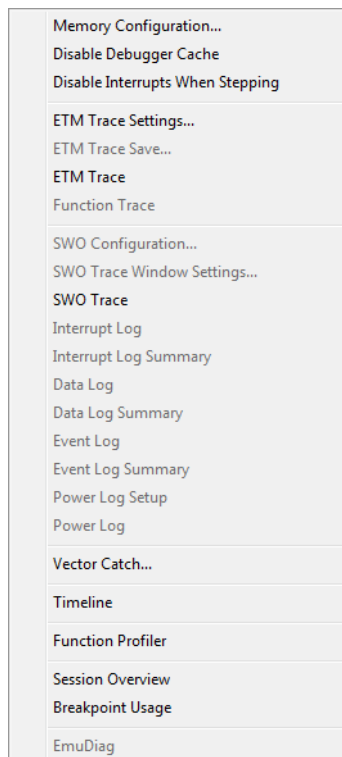
These commands are available on the menu:

Breakpoint Usage

Opens a window which lists all active breakpoints; see *Breakpoint Usage window*, page 139.

I-jet/JTAGjet menu

When you are using the C-SPY I-jet/JTAGjet driver, the **I-jet/JTAGjet** menu is added to the menu bar.



Menu commands

These commands are available on the menu:

Memory Configuration

Displays a dialog box; see *Memory Configuration dialog box, in C-SPY hardware debugger drivers*, page 191.

Disable Debugger Cache

Disables memory caching and memory range checking in C-SPY.

Normally, C-SPY uses the memory range information in the **Memory Configuration** dialog box both to restrict access to certain parts of target memory and to cache target memory contents for improved C-SPY performance. Under certain rare circumstances, this is not appropriate, and you can choose **Disable Debugger Cache** to turn off the caching and memory range checking completely. All accesses from C-SPY will then result in corresponding accesses to the target system. Some of those circumstances are:

- When memory is remapped at runtime and cannot be specified as a fixed set of ranges.
- When the memory range setup is incorrect or incomplete.

Disable Interrupts When Stepping

Ensures that only the stepped instructions will be executed. Interrupts will not be executed. This command can be used when not running at full speed and some interrupts interfere with the debugging process.

ETM Trace Settings

Displays a dialog box; see *ETM Trace Settings dialog box*, page 208.

ETM Trace Save

Displays a dialog box; see *Trace Save dialog box*, page 253.

ETM Trace

Opens the ETM Trace window; see *Trace window*, page 218.

Function Trace

Opens a window; see *Function Trace window*, page 223.

SWO Configuration

Displays a dialog box; see *SWO Configuration dialog box*, page 214.

This menu command is only available when the SWD/SWO interface is used.

SWO Trace Window Settings

Displays a dialog box; see *SWO Trace Window Settings dialog box*, page 212.

SWO Trace

Opens the SWO Trace window to display the collected trace data; see *Trace window*, page 218.

This menu command is only available when the SWD/SWO interface is used.

Interrupt Log

Opens a window; see *Interrupt Log window*, page 357.

This menu command is only available when the SWD/SWO interface is used.

Interrupt Log Summary

Opens a window; see *Interrupt Log Summary window*, page 361.

This menu command is only available when the SWD/SWO interface is used.

Data Log

Opens a window; see *Data Log window*, page 117.

This menu command is only available when the SWD/SWO interface is used.

Data Log Summary

Opens a window; see *Data Log Summary window*, page 119.

This menu command is only available when the SWD/SWO interface is used.

Event Log

Opens a window; see *Event Log window*, page 121.

Event Log Summary

Opens a window; see *Event Log Summary window*, page 123.

Power Log Setup

Opens a window; see *Power Log Setup window*, page 285.

Power Log

Opens a window; see *Power Log window*, page 287.

Vector Catch

Displays a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see *Vector Catch dialog box*, page 156. Note that this command is not available for all ARM cores.

Timeline

Opens a window; see *Timeline window*, page 224.

This menu command is only available when the SWD/SWO interface is used.

Function Profiler

Opens a window which shows timing information for the functions; see *Function Profiler window*, page 263.

Session Overview

Displays a window that lists information about the debug session, such as details about project settings, session settings, and the session state. To save the contents of the window to a file, choose **Save As** from the context menu.

Breakpoint Usage

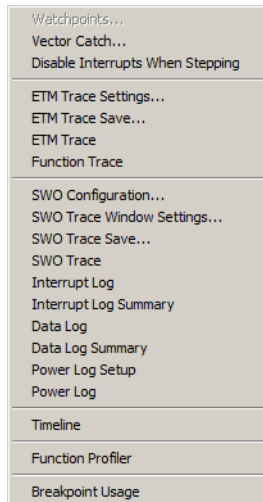
Opens a window which lists all active breakpoints; see *Breakpoint Usage window*, page 139.

EmuDiag

Starts the **EmuDiag** application where you can diagnose the connection between the host computer, the probe, and the board.

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:

Watchpoints

Displays a dialog box for setting watchpoints, see *Code breakpoints dialog box*, page 140.

Vector Catch

Displays a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see *Vector Catch dialog box*, page 156. Note that this command is not available for all ARM cores.

Disable Interrupts When Stepping

Ensures that only the stepped instructions will be executed. Interrupts will not be executed. This command can be used when not running at full speed and some interrupts interfere with the debugging process.

ETM Trace Settings

Displays a dialog box to configure ETM trace data generation and collection; see *ETM Trace Settings dialog box (J-Link/J-Trace)*, page 210.

This menu command is only available when using either ETM or J-Link with ETB.

ETM Trace Save

Displays a dialog box to save the collected trace data to a file; see *Trace Save dialog box*, page 253.

This menu command is only available when using either ETM or J-Link with ETB.

ETM Trace

Opens the ETM Trace window to display the collected trace data; see *Trace window*, page 218.

This menu command is only available when using either ETM or J-Link with ETB.

Function Trace

Opens a window which displays the trace data for function calls and function returns; see *Function Trace window*, page 223.

This menu command is only available when using either ETM or J-Link with ETB.

SWO Configuration

Displays a dialog box; see *SWO Configuration dialog box*, page 214.

This menu command is only available when the SWD/SWO interface is used.

SWO Trace Window Settings

Displays a dialog box; see *SWO Trace Window Settings dialog box*, page 212.

This menu command is only available when the SWD/SWO interface is used.

SWO Trace Save

Displays a dialog box to save the collected trace data to a file; see *Trace Save dialog box*, page 253.

This menu command is only available when the SWD/SWO interface is used.

SWO Trace

Opens the SWO Trace window to display the collected trace data; see *Trace window*, page 218.

This menu command is only available when the SWD/SWO interface is used.

Interrupt Log

Opens a window; see *Interrupt Log window*, page 357.

This menu command is only available when the SWD/SWO interface is used.

Interrupt Log Summary

Opens a window; see *Interrupt Log Summary window*, page 361.

This menu command is only available when the SWD/SWO interface is used.

Data Log

Opens a window; see *Data Log window*, page 117.

This menu command is only available when the SWD/SWO interface is used.

Data Log Summary

Opens a window; see *Data Log Summary window*, page 119.

This menu command is only available when the SWD/SWO interface is used.

Power Log Setup

Opens a window; see *Power Log Setup window*, page 285.

Power Log

Opens a window; see *Power Log window*, page 287.

Timeline

Opens a window; see *Timeline window*, page 224.

This menu command is available when using ETM or SWD/SWO.

Function Profiler

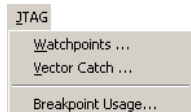
Opens a window which shows timing information for the functions; see *Function Profiler window*, page 263.

Breakpoint Usage

Opens a window which lists all active breakpoints; see *Breakpoint Usage window*, page 139.

Macraigor JTAG menu

When you are using the C-SPY Macraigor driver, the **JTAG** menu is added to the menu bar.

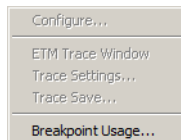


These commands are available on the menu:

- | | |
|-------------------------|---|
| Watchpoints | Opens a dialog box for setting watchpoints, see <i>Code breakpoints dialog box</i> , page 140. |
| Vector Catch | Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see <i>Vector Catch dialog box</i> , page 156. Note that this command is not available for all ARM cores. |
| Breakpoint Usage | Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 139. |

RDI menu

When you are using the C-SPY RDI driver, the **RDI** menu is added to the menu bar.



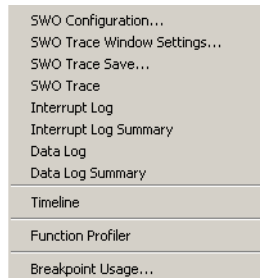
These commands are available on the menu:

- | | |
|-------------------------|---|
| Configure | Opens a dialog box that originates from the RDI driver vendor. For information about details in this dialog box, refer to the driver documentation. |
| Trace Settings | Displays a dialog box to configure the ETM trace; see <i>ETM Trace Settings dialog box (J-Link/J-Trace)</i> , page 210. |
| Trace Save | Displays a dialog box to save the collected trace data to a file; see <i>Trace Save dialog box</i> , page 253. |
| Breakpoint Usage | Opens a window which lists all active breakpoints; see <i>Breakpoint Usage window</i> , page 139. |

Note: To get the default settings in the configuration dialog box, it is for some RDI drivers necessary to just open and close the dialog box even though you do not need any specific settings for your project.

ST-LINK menu

When you are using the C-SPY ST-LINK driver, the **ST-LINK** menu is added to the menu bar.



These commands are available on the menu:

- | | |
|---|--|
| SWO Configuration ¹ | Displays a dialog box; see <i>SWO Configuration dialog box</i> , page 214. |
| SWO Trace Window Settings ¹ | Displays a dialog box; see <i>SWO Trace Window Settings dialog box</i> , page 212. |
| SWO Trace Save ¹ | Displays a dialog box to save the collected trace data to a file; see <i>Trace Save dialog box</i> , page 253. |
| SWO Trace ¹ | Opens the SWO Trace window to display the collected trace data; see <i>Trace window</i> , page 218. |
| Interrupt Log ¹ | Opens a window; see <i>Interrupt Log window</i> , page 357. |
| Interrupt Log Summary ¹ | Opens a window; see <i>Interrupt Log Summary window</i> , page 361. |
| Data Log ¹ | Opens a window; see <i>Data Log window</i> , page 117. |
| Data Log Summary ¹ | Opens a window; see <i>Data Log Summary window</i> , page 119. |
| Timeline ² | Opens a window; see <i>Timeline window</i> , page 224. |
| Function Profiler | Opens a window which shows timing information for the functions; see <i>Function Profiler window</i> , page 263. |

Breakpoint Usage Opens a window which lists all active breakpoints; see *Breakpoint Usage window*, page 139.

1 Only available when the SWD/SWO interface is used.

2 Available when using either ETM or SWD/SWO.

TI Stellaris menu

When you are using the C-SPY TI Stellaris driver, the **TI Stellaris** menu is added to the menu bar.

A screenshot of a menu item labeled "Breakpoint Usage ...".

This command is available on the menu:

Breakpoint Usage Opens a window which lists all active breakpoints; see *Breakpoint Usage window*, page 139.

TI XDS menu

When you are using the C-SPY TI XDS driver, the **TI XDS** menu is added to the menu bar.

A screenshot of a menu item labeled "Breakpoint Usage ...".

This command is available on the menu:

Breakpoint Usage Opens a window which lists all active breakpoints; see *Breakpoint Usage window*, page 139.

Reference information on the C-SPY simulator

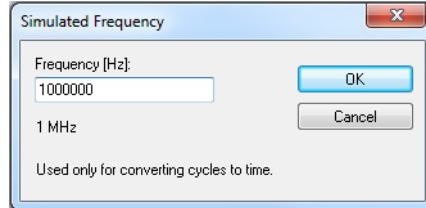
This section gives additional reference information the C-SPY simulator, reference information not provided elsewhere in this documentation.

Reference information about:

- *Simulated Frequency dialog box*, page 555

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.

Resolving problems

These topics are covered:

- No contact with the target hardware
- Slow stepping speed

Debugging using the C-SPY hardware debugger systems requires interaction between many systems, independent from each other. For this reason, setting up this debug system can be a complex task. If something goes wrong, it might be difficult to locate the cause of the problem.

For information about the current debug session, choose **Session Overview** from the driver menu. Note that this window might not be supported by the C-SPY driver you are using.

This section includes suggestions for resolving the most common problems that can occur when debugging with the C-SPY hardware debugger systems.

For problems concerning the operation of the evaluation board, refer to the documentation supplied with it, or contact your hardware distributor.

NO CONTACT WITH THE TARGET HARDWARE

There are several possible reasons for C-SPY to fail to establish contact with the target hardware. Do this:

- Check the communication devices on your host computer
- Verify that the cable is properly plugged in and not damaged or of the wrong type
- Make sure that the evaluation board is supplied with sufficient power
- Check that the correct options for communication have been specified in the IAR Embedded Workbench IDE.
- Check that the correct reset strategy is used.

Examine the linker configuration file to make sure that the application has not been linked to the wrong address.

SLOW STEPPING SPEED

If you find that the stepping speed is slow, these troubleshooting tips might speed up stepping:

- If you are using a hardware debugger system, keep track of how many hardware breakpoints that are used and make sure some of them are left for stepping.
Stepping in C-SPY is normally performed using breakpoints. When C-SPY performs a step command, a breakpoint is set on the next statement and the application executes until it reaches this breakpoint. If you are using a hardware debugger system, the number of hardware breakpoints—typically used for setting a stepping breakpoint in code that is located in flash/ROM memory—is limited. If you, for example, step into a C `switch` statement, breakpoints are set on each branch; this might consume several hardware breakpoints. If the number of available hardware breakpoints is exceeded, C-SPY switches into single stepping on assembly level, which can be very slow.
For more information, see *Breakpoints in the C-SPY hardware debugger drivers*, page 129 and *Breakpoint consumers*, page 129.
- Disable trace data collection, using the **Enable/Disable** button in both the **Trace** and the **Function Profiling** windows. Trace data collection might slow down stepping because the collected trace data is processed after each step. Note that it is not sufficient to just close the corresponding windows to disable trace data collection.
- Choose to view only a limited selection of SFR registers. You can choose between two alternatives. Either type `#SFR_name` (where `SFR_name` reflects the name of the SFR you want to monitor) in the **Watch** window, or create your own filter for displaying a limited group of SFRs in the **Register** window. Displaying many SFR registers might slow down stepping because all registers must be read from the

hardware after each step. See *Defining application-specific register groups*, page 165.

- 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.

A

Abort (Report Assert option) 88
 absolute location, specifying for a breakpoint 157
 Access Type (Data breakpoints option) 143
 Access type (Edit Memory Access option) 191
 Access (Edit SFR option) 187
 accesses outside the bounds of arrays and
 other objects, detecting 305
 Actual (SWO clock setting) 216
 Adaptive (JTAG/SWD speed setting) 518, 526
 Add to Watch Window (Symbolic Memory window context
 menu) 176
 Add (SFR Setup window context menu) 185
 Address Bus Pattern (Address setting) 143
 Address Range (Find in Trace option) 252
 Address (Edit SFR option) 186
 Address (JTAG Watchpoints option) 143
 Allow ETB (I-jet/JTAGjet Trace option) 521
 Allow hardware reset (RDI option) 532
 Ambiguous symbol (Resolve Symbol Ambiguity option) . 116
 Any Size (Data setting) 143
 Any (Access Type setting) 143
 Any (Extern setting) 144
 Any (Mode setting) 144
 Append to file (Trace Save option) 253
 application, built outside the IDE 52
 assembler labels, viewing 96
 assembler source code, fine-tuning 255
 assembler symbols, using in C-SPY expressions 93
 assembler variables, viewing 96
 assumptions, programming experience 25
 Attach to program (debugger option) 500
 Auto Scroll (Timeline window context menu) 230
 Auto window 101
 Auto (Default breakpoint type setting) 154
 Auto (JTAG/SWD speed setting) 518, 526
 Auto (Probe config setting) 510, 517
 Auto (SWO protocol setting) 522

Auto (SWO setting) 527
 Autodetect (SWO clock setting) 216
 Autodetect (SWO prescaler setting) 217
 Autostep settings dialog box 89
 Autostep (Debug menu) 63

B

--backend (C-SPY command line option) 449
 backtrace information
 generated by compiler 76
 viewing in Call Stack window 82
 batch mode, using C-SPY in 439
 Baud rate (Macraigor option) 531
 Baud rate (Serial port settings option) 506, 513
 --BE32 (C-SPY command line option) 442
 --BE8 (C-SPY command line option) 442
 Big Endian (Memory window context menu) 170
 bit loss or undefined behavior when shifting, detecting . 303
 blocks, in C-SPY macros 375
 bold style, in this guide 30
 bounds of arrays and other objects, detect
 accesses outside 305
 --bounds_table_size (linker option) 324
 Break Condition (JTAG Watchpoints option) 144
 Break on Throw (Debug menu) 64
 Break on Uncaught Exception (Debug menu) 64
 Break (Debug menu) 63
 breakpoint condition, example 135–136
 Breakpoint type (Code breakpoints option) 140
 Breakpoint Usage window 139
 Breakpoint Usage (CMSIS-DAP menu) 545
 Breakpoint Usage (GDB Server menu) 545
 Breakpoint Usage (I-jet/JTAGjet menu) 549
 Breakpoint Usage (J-Link menu) 551
 Breakpoint Usage (Macraigor JTAG menu) 552
 Breakpoint Usage (RDI menu) 552
 Breakpoint Usage (Simulator menu) 543
 Breakpoint Usage (ST-LINK menu) 554

Breakpoint Usage (TI Stellaris menu)	554
Breakpoint Usage (TI XDS menu)	554
breakpoints	
code, example	416
connecting a C-SPY macro	370
consumers of	129
data	146
data log	150–151
description of	126
disabling used by Stack window	130
icons for in the IDE	128
in Memory window	132
listing all	139
profiling source	256, 264
reasons for using	125
setting	
in memory window	132
using system macros	133
using the dialog box	131
single-stepping if not available	50
toggling	131
types of	126
useful tips	135
Breakpoints dialog box	
Code	140
Data	146
Data Log	150–151
Immediate	155
Log	145
Trace Filter (J-Link)	247
Trace Start	235, 237, 239, 241–242
Trace Stop	236
Breakpoints options (C-SPY options)	153
Breakpoints window	137
Broadcast all branches (ETM Trace Settings option)	211
Browse (Trace toolbar)	218
Buffer limit (I-jet/JTAGjet Trace option)	521
byte order, setting in Memory window	169
Byte (Data setting)	143

C

C function information, in C-SPY	76
C symbols, using in C-SPY expressions	93
C variables, using in C-SPY expressions	93
Cache type (Edit Memory Range option)	195
call chain, displaying in C-SPY	76
Call stack information	76
Call Stack window	82
for backtrace information	76
Call Stack (Timeline window context menu)	231
__cancelAllInterrupts (C-SPY system macro)	383
__cancelInterrupt (C-SPY system macro)	384
Catch exceptions (Breakpoints option)	154
Catch exceptions (RDI option)	533
Chain (Break Condition setting)	145
checked heap, using	295
Clear All (Debug Log window context menu)	87
Clear trace data (Trace toolbar)	218
Clear (Interrupt Log window context menu)	360, 363
Clear (Power Log window context menu)	288
__clearBreak (C-SPY system macro)	384
Clock setup (J-Link/J-Trace option)	527
Clock setup (ST-LINK option)	535
Clock setup (TI XDS option)	538
__closeFile (C-SPY system macro)	384
CMSIS-DAP communication problem	511
CMSIS-DAP (C-SPY driver), menu	544
code breakpoints	
overview	126
toggling	131
Code Coverage window	270
Code Coverage (Disassembly window context menu)	80
--code_coverage_file (C-SPY command line option)	449
code, covering execution of	270
command line options	449
typographic convention	29
command prompt icon, in this guide	30
communication problem, CMSIS-DAP	511

- communication problem, I-jet 518
- communication problem, J-Link 526
- communication problem, ST-LINK 534
- Communication (Angel option) 506
- Communication (J-Link/J-Trace option) 528
- computer style, typographic convention 29
- conditional statements, in C-SPY macros 374
- Configure (RDI menu) 552
- Connect during reset (Reset setting) 508, 515, 523
- context menu, in windows 96
- conventions, used in this guide 29
- Copy Window Contents (Disassembly window context menu) 81
- Copy (Debug Log window context menu) 86
- copyright notice 2
- Core and peripherals (Reset setting) 523
- Core (Cores window) 340
- Core (Reset setting) 507, 514, 523
- cores
 - debugging multiple 335
 - inspecting state of 339
- Cores toolbar 341
- Cores window 339
- CPI (Generate setting) 213
- cpu (C-SPY command line option) 442
- CPU clock (I-jet/JTAGjet Trace option) 522
- CPU clock (SWO Configuration option) 216
- CPU clock (SWO setting) 527
- CPU number on target (Explicit probe configuration setting) 511, 519
- cspybat 439
 - reading options from file (-f) 465
- current position, in C-SPY Disassembly window 79
- cursor, in C-SPY Disassembly window 79
- Custom (Reset setting) 508, 515
- Cycle accurate tracing (ETM Trace Settings option) 211
- cycles (C-SPY command line option) 450
- Cycles (Cores window) 340
- C-RUN
 - creating rules for messages 299
 - detecting various runtime errors 299
 - getting started 297
 - in non-interactive mode 297
 - in the IDE 295
 - requirements for 296
 - setting options for 318
 - using 297
 - using the checked heap 295
- C-RUN Messages Rules window (View menu) 322
- C-RUN Messages window (View menu) 320
- C-RUN runtime error checking 293
- C-SPY
 - batch mode, using in 439
 - debugger systems, overview of 37
 - differences between drivers 39
 - environment overview 33
 - plugin modules, loading 51
 - setting up 50–51
 - starting the debugger 52
- C-SPY drivers
 - overview 39
 - specifying 499
 - types of 38
- C-SPY expressions 92
 - evaluating, using Macro Quicklaunch window 437
 - evaluating, using Quick Watch window 112
 - in C-SPY macros 374
 - Tooltip watch, using 91
 - Watch window, using 91
- C-SPY hardware debugger driver
 - extending functionality of 57
- C-SPY hardware drivers, hardware installation 44
- C-SPY macros
 - blocks 375
 - conditional statements 374
 - C-SPY expressions 374
 - examples 367
 - checking status of register 370
 - creating a log macro 371

execUserPreload, using	56
remapping memory before download	56
executing	367
connecting to a breakpoint	370
using Quick Watch	370
using setup macro and setup file	369
functions	94, 372
loop statements	375
macro statements	374
parameters	373
setup macro file	366
executing	369
setup macro functions	366
summary	377
system macros, summary of	380
using	365
variables	94, 373
C-SPY options	
Extra Options	503
Images	501
Multicore	504
Plugins	502
Setup	499
C-SPYLink	38
C-STAT for static analysis, documentation for	28
C++ exceptions	
debugging	64
single stepping	72
C++ terminology	29

D

data breakpoints, overview	127
Data Bus Pattern (Data setting)	144
Data Coverage (Memory window context menu)	170
data coverage, in Memory window	168
data log breakpoints, overview	127
Data Log Events (SWO Configuration option)	215
Data Log Summary window	119
Data Log Summary (I-jet/JTAGjet menu)	548
Data Log Summary (J-Link menu)	551
Data Log Summary (Simulator menu)	543
Data Log Summary (ST-LINK menu)	553
Data Log window	117
Data Log (I-jet/JTAGjet menu)	548
Data Log (J-Link menu)	551
Data Log (Simulator menu)	542
Data Log (ST-LINK menu)	553
Data Log (Timeline window context menu)	231
Data value + exact addr (Data Log Events setting)	215
Data (JTAG Watchpoints option)	143
DCC (Debug Communications Channel)	84, 107
ddf (filename extension), selecting a file	51
Debug handler address (Macraigor option)	531
Debug Log window	86
Debug menu (C-SPY main window)	62
Debug Probe Selection dialog box	44, 455–458
Debug (Report Assert option)	88
--debugfile (cspybat option)	450
debugger concepts, definitions of	36
debugger drivers	
<i>See</i> C-SPY drivers	39
simulator	41
Debugger Macros window	435
debugger system overview	37
debugging projects	
externally built applications	52
loading multiple images	53
debugging, RTOS awareness	35
--debug_heap (linker option)	325
Default breakpoint type (Breakpoints option)	153
default_no_bounds (pragma directive)	327
define_without_bounds (pragma directive)	328
define_with_bounds (pragma directive)	327
__delay (C-SPY system macro)	385
Delay after (CMSIS-DAP option)	509
Delay after (I-jet/JTAGjet option)	516
Delay (Autostep Settings option)	89

- Delete (Breakpoints window context menu) 138
 - Delete (SFR Setup window context menu) 185
 - Delete/revert All Custom SFRs (SFR Setup window context menu) 185
 - Description (Edit Interrupt option) 353
 - description (interrupt property) 353
 - device (C-SPY command line option) 451
 - Device description file (debugger option) 499
 - device description files 51
 - definition of 54
 - modifying 54
 - specifying interrupts 408
 - Device Support Module 57
 - Disable All (Breakpoints window context menu) 138
 - Disable Debugger Cache (CMSIS-DAP menu) 544
 - Disable Debugger Cache (I-jet/JTAGjet menu) 546
 - Disable Interrupts When Stepping (CMSIS-DAP menu) 544
 - Disable Interrupts When Stepping (I-jet/JTAGjet menu) 547
 - Disable Interrupts When Stepping (J-Link menu) 550
 - Disable (Breakpoints window context menu) 138
 - Disabled (Reset setting) 507, 514
 - __disableInterrupts (C-SPY system macro) 385
 - disable_check (pragma directive) 328
 - disable_interrupts (C-SPY command line option) 451
 - Disassemble in ARM mode (Disassembly menu) 65
 - Disassemble in Auto mode (Disassembly menu) 65
 - Disassemble in Current processor mode (Disassembly menu) 65
 - Disassemble in Thumb mode (Disassembly menu) 65
 - Disassembly menu (C-SPY main window) 65
 - Disassembly window 78
 - context menu 80, 341
 - disclaimer 2
 - Divider (PC Sampling setting) 215
 - division by zero, detecting 304
 - DLIB
 - consuming breakpoints 129
 - documentation 28
 - naming convention 30
 - do (macro statement) 375
 - document conventions 29
 - documentation
 - overview of guides 27
 - overview of this guide 26
 - this guide 25
 - download_only (C-SPY command line option) 451
 - Driver (debugger option) 499
 - __driverType (C-SPY system macro) 385
 - drv_attach_to_program (C-SPY command line option) 443
 - drv_catch_exceptions (C-SPY command line option) 452
 - drv_communication (C-SPY command line option) 453
 - drv_communication_log (C-SPY command line option) 459
 - drv_default_breakpoint (C-SPY command line option) 459
 - drv_interface (C-SPY command line option) 460
 - drv_interface_speed (C-SPY command line option) 461
 - drv_reset_to_cpu_start (C-SPY command line option) 462
 - drv_restore_breakpoints (C-SPY command line option) 462
 - drv_suppress_download (C-SPY command line option) 443
 - drv_swo_clock_setup (C-SPY command line option) 463
 - drv_vector_table_base (C-SPY command line option) 464
 - drv_verify_download (C-SPY command line option) 443
 - Duration (CMSIS-DAP option) 509
 - Duration (I-jet/JTAGjet option) 516
- ## E
- Edit Expressions (Trace toolbar) 219
 - Edit Interrupt dialog box 353
 - Edit Memory Range dialog box 186, 194
 - Edit Memory Range dialog box (C-SPY simulator) 190
 - Edit Nickname (Debug Probe Selection dialog box) 44
 - Edit Settings (Trace toolbar) 219
 - Edit (Breakpoints window context menu) 138
 - Edit (SFR Setup window context menu) 185
 - edition, of this guide 2
 - Embedded C++ Technical Committee 29
 - EmbeddedICE macrocell 127
 - EmuDiag (I-jet/JTAGjet menu) 549

Emulator (ST-LINK option)	533	Event Log Summary (I-jet/JTAGjet menu)	548
Emulator (TI XDS option)	537	Event Log window	121
__emulatorSpeed (C-SPY system macro)	386	Event Log (I-jet/JTAGjet menu)	548
__emulatorStatusCheckOnRead (C-SPY system macro)	387	Event Log (Timeline window context menu)	231
Enable All (Breakpoints window context menu)	138	examples	
Enable interrupt simulation (Interrupt Setup option)	351	C-SPY macros	367
Enable Log File (Log File option)	87	interrupts	
Enable multicore master mode (debugger option)	504	interrupt logging	350
Enable runtime checking (C-RUN option)	318, 321	timer	348
Enable (Breakpoints window context menu)	138	macros	
Enable (Interrupt Log window context menu)	360, 363	checking status of register	370
Enable (Power Log window context menu)	288	creating a log macro	371
Enable (Timeline window context menu)	231	using Quick Watch	370
Enabled ports (ITM Stimulus Ports setting)	217	performing tasks and continue execution	136
__enableInterrupts (C-SPY system macro)	388	tracing incorrect function arguments	135
Enable/Disable Breakpoint (Call		EXC (Generate setting)	213
Stack window context menu)	83	execUserExecutionStarted (C-SPY setup macro)	378
Enable/Disable Breakpoint (Disassembly window context		execUserExecutionStopped (C-SPY setup macro)	378
menu)	81	execUserExit (C-SPY setup macro)	380
Enable/Disable (Trace toolbar)	218	execUserFlashExit (C-SPY setup macro)	380
--endian (C-SPY command line option)	443	execUserFlashInit (C-SPY setup macro)	378
endianness. <i>See</i> byte order		execUserFlashReset (C-SPY setup macro)	379
Enter Location dialog box	157	execUserPreload (C-SPY setup macro)	377
ETB trace	200	execUserPreReset (C-SPY setup macro)	379
ETM trace	200	execUserReset (C-SPY setup macro)	379
ETM Trace Save (CMSIS-DAP menu)	545	execUserSetup (C-SPY setup macro)	378
ETM Trace Save (I-jet/JTAGjet menu)	547	executed code, covering	270
ETM Trace Save (J-Link menu)	550	execution history, tracing	206
ETM Trace Settings dialog box	208, 210	Execution state (Cores window)	340
ETM Trace Settings (CMSIS-DAP menu)	544	Explicit (Probe config setting)	510, 517
ETM Trace Settings (I-jet/JTAGjet menu)	547	expressions. <i>See</i> C-SPY expressions	
ETM Trace Settings (J-Link menu)	550	Extend to cover requested range (Trigger range setting)	152
ETM Trace (CMSIS-DAP menu)	545	range (Trigger range setting)	148, 238, 241, 243, 246, 248
ETM Trace (I-jet/JTAGjet menu)	547	extended command line file, for cspyat	465
ETM Trace (J-Link menu)	550	Extern (JTAG Watchpoints option)	144
ETM/ETB (J-Link/J-Trace option)	527	Extra Options, for C-SPY	503
__evaluate (C-SPY system macro)	388		
Evaluate Now (Macro Quicklaunch			
window context menu)	438		
Event Log Summary window	123		

- ## F
- f (cspybat option) 465
 - Factory ranges (Memory Configuration option) 192
 - File format (Memory Save option) 171
 - file types
 - device description, specifying in IDE 51
 - macro 51, 499
 - File (Trace Save option) 253
 - filename extensions
 - ddf, selecting device description file 51
 - mac, using macro file 51
 - Filename (Memory Restore option) 172
 - Filename (Memory Save option) 172
 - Fill dialog box 173
 - __writeMemory8 (C-SPY system macro) 389
 - __writeMemory16 (C-SPY system macro) 389
 - __writeMemory32 (C-SPY system macro) 390
 - Find in Trace dialog box 251
 - Find in Trace window 252
 - Find (Memory window context menu) 170
 - Find (Trace toolbar) 219
 - first activation time (interrupt property)
 - definition of 345
 - First activation (Edit Interrupt option) 354
 - Fixed (JTAG/SWD speed setting) 526
 - flash loader
 - parameters to control 494
 - specifying the path to 494
 - using 489
 - Flash Loader Overview dialog box 491
 - flash memory, load library module to 404
 - flash_loader (C-SPY command line option) 465
 - FOLD (Generate setting) 213
 - for (macro statement) 375
 - Force (SWO Trace Window Settings option) 212
 - Forced Interrupt window 354
 - Forced Interrupts (Simulator menu) 543
 - fpu (C-SPY command line option) 443
 - From file (Probe config setting) 510, 517
 - Function Profiler window 263
 - Function Profiler (CMSIS-DAP menu) 545
 - Function Profiler (I-jet/JTAGjet menu) 548
 - Function Profiler (J-Link menu) 551
 - Function Profiler (Simulator menu) 542
 - Function Profiler (ST-LINK menu) 553
 - Function Trace window 223
 - Function Trace (CMSIS-DAP menu) 545
 - Function Trace (I-jet/JTAGjet menu) 547
 - Function Trace (J-Link menu) 550
 - Function Trace (Simulator menu) 542
 - functions
 - call stack information for 76
 - C-SPY running to when starting 50, 499
 - most time spent in, locating 255
- ## G
- GDB Server (C-SPY driver), menu 545
 - __gdbserver_exec_command (C-SPY system macro) 391
 - gdbserv_exec_command (C-SPY command line option) 466
 - Generate (SWO Trace Window Settings option) 213
 - generate_entries_without_bounds (compiler option) 325
 - generate_entry__without_bounds (pragma directive) 328
 - Get Example Projects dialog box 69
 - __getSelectedCore (C-SPY system macro) 392
 - __getTracePortSize (C-SPY system macro) 392
 - Go to Source (Breakpoints window context menu) 138
 - Go to Source (Call Stack window context menu) 83
 - Go To Source (Timeline window context menu) 232–233
 - Go (Debug menu) 62, 75
 - Graphical bar (Memory Configuration dialog box) 193
- ## H
- Halfword (Data setting) 143
 - Halt after bootloader (Reset setting) 524
 - Halt before bootloader (Reset setting) 524

Hardware reset (Macraigor option)	531
hardware setup, power consumption because of	280
Hardware (Default breakpoint type setting)	154
Hardware (Reset setting)	507, 514
Hardware, Atmel AT91SAM7 (Reset setting)	525
Hardware, halt after delay (ms) (Reset setting)	524
Hardware, halt at 0 (Reset setting)	525
Hardware, halt using Breakpoint (Reset setting)	525
Hardware, halt using DBGRQ (Reset setting)	525
Hardware, NXP LPC (Reset setting)	525
__hasDAPRegs (C-SPY system macro)	393
heap	295
heap integrity violations, detecting	314
heap memory leaks, detecting	312
heap usage error, detecting	311
Hexadecimal (Timeline window context menu)	232
highlighting, in C-SPY	76
Hold time (Edit Interrupt option)	354
hold time (interrupt property), definition of	345
__hwJetResetWithStrategy (C-SPY system macro)	393
__hwReset (C-SPY system macro)	394
__hwResetRunToBp (C-SPY system macro)	394
__hwResetWithStrategy (C-SPY system macro)	396
__hwRunToBreakpoint (C-SPY system macro)	396

I

IAR debugger driver plugin (debugger option)	539
icons, in this guide	30
if else (macro statement)	375
if (macro statement)	374
Ignore (Report Assert option)	88
--ignore_uninstrumented_pointers (compiler option)	326
--ignore_uninstrumented_pointers (linker option)	326
illegal memory accesses, checking for	164
Images window	67
Images, loading multiple	501
immediate breakpoints, overview	127
implicit or explicit integer conversion, detecting	299

In use by (Data Log Events setting)	215
In use by (PC Sampling setting)	215
Include (Log File option)	87
Index Range (Trace Save option)	253
Input Mode dialog box	85
input, special characters in Terminal I/O window	85
insert checks for (C-RUN option)	320
installation directory	29
Instruction Profiling (Disassembly window context menu)	80
integer conversion, detect implicit or explicit	299
Intel-extended, C-SPY output format	38
Interface (CMSIS-DAP option)	510
Interface (I-jet/JTAGjet option)	517
Interface (J-Link/J-Trace option)	528
Interface (Macraigor option)	530
Interface (ST-LINK option)	534
Interface (TI Stellaris option)	536
Interface (TI XDS option)	537
interference, power consumption because of	281
interrupt handling, power consumption during	279
Interrupt Log Summary window	361
Interrupt Log Summary (I-jet/JTAGjet menu)	548
Interrupt Log Summary (J-Link menu)	551
Interrupt Log Summary (Simulator menu)	543
Interrupt Log Summary (ST-LINK menu)	553
Interrupt Log window	357
Interrupt Log (I-jet/JTAGjet menu)	547
Interrupt Log (J-Link menu)	551
Interrupt Log (Simulator menu)	543
Interrupt Log (ST-LINK menu)	553
Interrupt Log (SWO Configuration option)	216
Interrupt Logs (Force setting)	213
Interrupt Setup dialog box	351
Interrupt Setup (Simulator menu)	543
Interrupt Status window	355
interrupt system, using device description file	347
Interrupt (Edit Interrupt option)	353
Interrupt (Timeline window context menu)	231

interrupts

adapting C-SPY system for target hardware	347
simulated, introduction to	343
timer, example	348
using system macros	346
__isBatchMode (C-SPY system macro)	397
italic style, in this guide	29–30
ITM Log (Force setting)	213
ITM Stimulus Ports (SWO Configuration option)	217
I-jet communication problem	518
I-jet JTAG interface	514
I-jet Trace	514
I-jet/JTAGjet (C-SPY driver), menu	546

J

--jet_board_cfg (C-SPY command line option)	466
--jet_board_did (C-SPY command line option)	466
--jet_cpu_clock (C-SPY command line option)	467
--jet_ir_length (C-SPY command line option)	468
--jet_power_from_probe (C-SPY command line option)	468
--jet_probe (C-SPY command line option)	469
--jet_script_file (C-SPY command line option)	469
--jet_standard_reset (C-SPY command line option)	470
--jet_startup_connection_timeout (C-SPY command line option)	471
--jet_swo_on_d0 (C-SPY command line option)	471
--jet_swo_prescaler (C-SPY command line option)	472
--jet_swo_protocol (C-SPY command line option)	472
--jet_tap_position (C-SPY command line option)	473
__jlinkExecCommand (C-SPY system macro)	398
--jlink_dcc_timeout (C-SPY command line option)	473
--jlink_device_select (C-SPY command line option)	474
--jlink_exec_command (C-SPY command line option)	474
--jlink_initial_speed (C-SPY command line option)	474
--jlink_ir_length (C-SPY command line option)	475
--jlink_reset_strategy (C-SPY command line option)	475
--jlink_script_file (C-SPY command line option)	476
--jlink_trace_source (C-SPY command line option)	476

JTAG interfaces

I-jet	514
J-Link	523, 528
JTAG scan chain contains non-ARM devices (Explicit probe configuration setting)	519
JTAG scan chain with multiple targets (Macraigor option)	531
JTAG scan chain with multiple targets (J-Link/J-Trace option)	529
JTAG scan chain (CMSIS-DAP option)	511
JTAG scan chain (I-jet/JTAGjet option)	518
JTAG scan chain (J-Link/J-Trace option)	529
JTAG speed (Macraigor option)	530
JTAG Watchpoints dialog box	142
JTAG watchpoints, overview	127
JTAG (Interface setting)	517, 528, 530
__jtagCommand (C-SPY system macro)	398
__jtagCP15IsPresent (C-SPY system macro)	399
__jtagCP15ReadReg (C-SPY system macro)	399
__jtagCP15WriteReg (C-SPY system macro)	399
__jtagData (C-SPY system macro)	400
__jtagRawRead (C-SPY system macro)	400
__jtagRawSync (C-SPY system macro)	401
__jtagRawWrite (C-SPY system macro)	402
__jtagResetTRST (C-SPY system macro)	403
JTAG/SWD speed (CMSIS-DAP option)	510
JTAG/SWD speed (I-jet/JTAGjet option)	518
JTAG/SWD speed (J-Link/J-Trace option)	526
JTAG/SWD speed (ST-LINK option)	534
JTAG/SWD speed (TI Stellaris option)	536
JTAG/SWD speed (TI XDS option)	537
J-Link communication problem	526
J-Link JTAG interface	523, 528
J-Link (C-SPY driver), menu	549

L

labels (assembler), viewing	96
Leave on after debugging (Target power supply setting)	516
Length (Fill option)	173

library functions	
C-SPY support for using, plugin module	481
online help for	28
lightbulb icon, in this guide	30
Link condition (Trace Start option)	244
Link condition (Trace Stop option)	247, 249
linker options	
typographic convention	29
consuming breakpoints	129
Little Endian (Memory window context menu)	169
Live Watch window	106
__loadImage (C-SPY system macro)	403
loading multiple debug files, list currently loaded	67
loading multiple images	53
Locals window	102
log breakpoints, overview	126
Log communication (Angel option)	507, 512
Log communication (debugger option)	509, 529, 532–533, 535–536, 538–539
Log communication (IAR ROM-monitor option)	513
Log communication (I-jet/JTAGjet option)	517
Log File dialog box	87
Logging>Set Log file (Debug menu)	65
Logging>Set Terminal I/O Log file (Debug menu)	65
loop statements, in C-SPY macros	375
low-power mode, power consumption during	278
LSU (Generate setting)	213

M

mac (filename extension), using a macro file	51
Macraigor (C-SPY driver), menu	552
--macro (C-SPY command line option)	477
macro files, specifying	51, 499
Macro Quicklaunch window	437
Macro Registration window	433
macro statements	374
macros	
executing	367

using	365
Macros (Debug menu)	64
--macro-param (C-SPY command line option)	477
--mac_handler_address (C-SPY command line option)	478
--mac_jtag_device (C-SPY command line option)	478
--mac_multiple_targets (C-SPY command line option)	479
--mac_reset_pulls_reset (C-SPY command line option)	479
--mac_set_temp_reg_buffer (C-SPY command line option)	480
--mac_xscale_ir7 (C-SPY command line option)	480
main function, C-SPY running to when starting	50, 499
Manchester (SWO protocol setting)	522
Manufacturer RDI driver (RDI option)	532
--mapu (C-SPY command line option)	480
Mask (Address setting)	143
Mask (Data setting)	144
Mask (Match data setting)	149, 238, 240, 244, 247, 249
Match data (Data breakpoints option)	149
Match data (Trace Start option)	238, 240, 244
Match data (Trace Stop option)	246, 249
memory access checking	164
Memory access checking (Memory Access Setup option)	189
memory accesses, illegal	164
Memory Configuration dialog box	191
Memory Configuration dialog box (in C-SPY simulator)	187
Memory Configuration (CMSIS-DAP menu)	544
Memory Configuration (I-jet/JTAGjet menu)	546
Memory Configuration (Simulator menu)	542
Memory Fill (Memory window context menu)	170
Memory Restore dialog box	172
Memory Restore (Memory window context menu)	170
Memory Save dialog box	171
Memory Save (Memory window context menu)	170
Memory window	167
memory zones	163
__memoryRestore (C-SPY system macro)	404
__memorySave (C-SPY system macro)	405
Memory>Restore (Debug menu)	64
Memory>Save (Debug menu)	64
menu bar, C-SPY-specific	62

__messageBoxYesCancel (C-SPY system macro) 406
 __messageBoxYesNo (C-SPY system macro) 406
 migration, from earlier IAR compilers 28
 MISRA C
 documentation 28
 Mixed Mode (Disassembly window context menu) 81
 Mode (JTAG Watchpoints option) 144
 Motorola, C-SPY output format 38
 Move to PC (Disassembly window context menu) 80
 MTB trace. 200
 multicore debugging 335
 Multicore (C-SPY options) 504
 Multi-ICE interface. *See* RealView Multi-ICE interface
 Multi-target debug system (Explicit probe configuration setting) 511, 518

N

n MHz (JTAG/SWD speed setting) 518
 Name (Edit SFR option) 186
 naming conventions 30
 Navigate (Timeline window context menu) 230
 New Breakpoint (Breakpoints window context menu) 139
 Next Statement (Debug menu) 63
 Next Symbol (Symbolic Memory window context menu) 176
 Non User (Mode setting) 144
 Normal (Break Condition setting) 144
 Normal (Reset setting) 523
 Normal, disable watchdog (Reset setting) 524
 no_bounds (pragma directive) 329
 Number of cores (debugger option) 504

O

OCD interface device (Macraigor option) 530
 OP Fetch (Access Type setting) 143
 Open Setup Window (Power Log window context menu) 289
 __openFile (C-SPY system macro) 407
 Operation (Fill option) 173

operators, sizeof in C-SPY 94
 optimizations, effects on variables 95
 options
 in the IDE 497
 on the command line 449, 503
 Options (Stack window context menu) 180
 __orderInterrupt (C-SPY system macro) 408
 Originator (debugger option) 503
 overflow, signed or unsigned 301
 Override default .board file (debugger option) 501
 Override default (Probe configuration file setting) 511, 518
 Override project default (SWO Configuration option) 216
 Override project setting (SWO Configuration option) 216
 overriding the default stack setup 177

P

-p (C-SPY command line option) 481
 parameters
 list of passed to the flash loader 491
 tracing incorrect values of 76
 typographic convention 29
 part number, of this guide 2
 PC only (Data Log Events setting) 215
 PC samples (Force setting) 213
 PC Sampling (SWO Configuration option) 215
 PC (Cores window) 340
 PC + data value + base addr (Data Log Events setting) 215
 peripheral units
 debugging power consumption for 275
 detecting mistakenly unattended 279
 detecting unattended 279
 device-specific 54
 displayed in Register window 162
 in an event-driven system 279
 in C-SPY expressions 93
 initializing using setup macros 366

peripheral units, in Register window 162

Please select one symbol
(Resolve Symbol Ambiguity option) 116

--plugin (C-SPY command line option) 481

plugin modules (C-SPY). 38
 loading 51

Plugins (C-SPY options). 502

__popSimulatorInterruptExecutingStack (C-SPY
system macro). 409

pop-up menu. *See* context menu

Port (Macraigor option) 531

Port (Serial port settings option) 506, 513

power consumption, measuring. 256, 275

Power graph in Timeline window 290

Power Log Setup (I-jet/JTAGjet menu). 548

Power Log Setup (J-Link menu) 551

Power Log window 287

Power Log (I-jet/JTAGjet menu). 548

Power Log (J-Link menu) 551

Power Log (Timeline window context menu) 231

power sampling. 256

Power Setup window 285

Preceding bits (JTAG scan chain setting) 519, 529

prerequisites, programming experience. 25

Previous Symbol (Symbolic
Memory window context menu) 176

probability (interrupt property) 354
 definition of 345

Probability % (Edit Interrupt option) 354

Probe config (CMSIS-DAP option) 510

Probe config (I-jet/JTAGjet option). 517

Probe configuration file (CMSIS-DAP option). 511

Probe configuration file (I-jet/JTAGjet option) 518

--proc_stack_xxx (C-SPY command line option) 482

Profile Selection (Timeline window context menu) 233

profiling
 analyzing data 258
 on function level 258
 on instruction level. 260

profiling information, on functions and instructions 255

profiling sources
 breakpoints 256, 264
 sampling 256, 264
 trace (calls) 256, 264
 trace (flat) 256, 264

program execution
 breaking 126–127
 in C-SPY 71
 multiple cores in C-SPY 335

programming experience. 25

program. *See* application

projects, for debugging externally built applications. 52

PTM trace 200

publication date, of this guide. 2

Q

Quick Watch window 112
 executing C-SPY macros 370

R

RAM (Edit Memory Access option). 195

Range for (Viewing Range option) 234

Range (Break Condition setting). 144

Rate (PC Sampling setting). 215

RDI (C-SPY driver), menu 552
 --rdi_allow_hardware_reset
 (C-SPY command line option) 482
 --rdi_driver_dll (C-SPY command line option). 483
 --rdi_heartbeat (C-SPY command line option) 444
 --rdi_step_max_one (C-SPY command line option) 483

Read (Access Type setting). 143

__readAPReg (C-SPY system macro) 409

__readDPRReg (C-SPY system macro) 410

__readFile (C-SPY system macro) 411

__readFileByte (C-SPY system macro) 411

reading guidelines. 25

__readMemoryByte (C-SPY system macro). 412

__readMemory8 (C-SPY system macro) 412
 __readMemory16 (C-SPY system macro) 412
 __readMemory32 (C-SPY system macro) 413
 RealView Multi-ICE interface 532
 reference information, typographic convention 30
 Refresh (Debug menu) 64
 register groups 162
 predefined, enabling 181
 Register window 180
 registered trademarks 2
 __registerMacroFile (C-SPY system macro). 413
 registers, displayed in Register window 180
 Remove All (Macro Quicklaunch window
 context menu) 438
 Remove (Macro Quicklaunch window context menu) . . . 438
 Repeat interval (Edit Interrupt option) 354
 repeat interval (interrupt property), definition of. 345
 Replace (Memory window context menu) 170
 Report Assert dialog box 88
 Reset and halt after bootloader (Reset setting) 508, 515
 Reset by watchdog or reset register (Reset setting). . 508, 515
 Reset Pin (Reset setting) 523
 Reset (CMSIS-DAP option) 507
 Reset (Debug menu) 63
 Reset (I-jet/JTAGjet option) 514
 Reset (J-Link/J-Trace option) 523
 Reset (ST-Link option) 534
 Reset (TI XDS option) 537
 __resetFile (C-SPY system macro). 414
 --reset_style (C-SPY command line option) 483
 Resolve Source Ambiguity dialog box 158
 Restore software breakpoints at (Breakpoints option). . . 154
 Restore (Memory Restore option). 172
 __restoreSoftwareBreakpoints (C-SPY system macro). . . 414
 return (macro statement). 375
 ROM-monitor, definition of 38
 ROM/Flash (Edit Memory Access option) 195
 RTOS awareness debugging 35
 RTOS awareness (C-SPY plugin module). 36
 Run to Cursor (Call Stack window context menu) 83

Run to Cursor (Debug menu) 63
 Run to Cursor (Disassembly window context menu) 80
 Run to Cursor, command for executing. 76
 Run to (C-SPY option) 50, 499
 runtime checking, setting options for C-RUN 318
 runtime error checking 293
 getting started
 requirements for 297
 requirements for. 296
 using C-RUN 294
 --runtime_checking (compiler option) 326
 Run/Step/Stop affect all cores (Cores
 window
 context menu) 341
 Run/Step/Stop affect current core only (Cores window context
 menu) 341
 R/W (Access Type setting) 143

S

sampling, profiling source. 256, 264
 Save Custom SFRs (SFR Setup window context menu) . . 186
 Save to log file (context menu command). 360, 363
 Save to log file (Power Log window context menu) 289
 Save (Memory Save option) 172
 Save (Trace toolbar) 219
 Scale (Viewing Range option). 234
 Scan chain contains non-ARM devices
 (JTAG scan chain setting) 529
 Select All (Debug Log window context menu). 86
 Select Graphs (Timeline window context menu). 233
 Select plugins to load (debugger option). 502
 Select (Probe configuration file setting) 511, 518
 __selectCore (C-SPY system macro) 415
 --semihosting (C-SPY command line option) 484
 Send heartbeat (Angel option) 506
 Serial port settings (Angel option) 506
 Serial port settings (IAR ROM-monitor option) 513
 Session Overview (CMSIS-DAP menu) 545
 Session Overview (I-jet/JTAGjet menu) 548

Set Data Breakpoint (Memory window context menu) . . .	170	Show variables (Stack window context menu)	179
Set Data Log		signed or unsigned overflow, detecting	301
Breakpoint (Memory window context menu)	171	--silent (C-SPY command line option)	485
Set Next Statement (Debug menu)	64	Simulated FrequencyDlg	555
Set Next Statement (Disassembly window context menu) .	81	simulating interrupts, enabling/disabling	351
__setCodeBreak (C-SPY system macro)	415	Simulator menu	542
__setDataBreak (C-SPY system macro)	416	simulator, introduction	41
__setDataLogBreak (C-SPY system macro)	418	Size (Edit SFR option)	187
__setLogBreak (C-SPY system macro)	419	Size (Timeline window context menu)	232
__setSimBreak (C-SPY system macro)	420	Size (Trace Start option)	238, 240, 243, 245, 248
__setTraceStartBreak (C-SPY system macro)	421	sizeof	94
__setTraceStopBreak (C-SPY system macro)	423	SLEEP (Generate setting)	213
setup macro file, registering	51	software delay, power consumption during	277
setup macro functions	366	Software (Default breakpoint type setting)	154
reserved names.	377	Software (Reset setting)	507, 514, 525
Setup macros (debugger option)	499	Software, Analog devices (Reset setting)	525
Setup (C-SPY options)	499	Solid Graph (Timeline window context menu)	232
SFR		__sourcePosition (C-SPY system macro)	425
in Register window	181	special function registers (SFR)	
using as assembler symbols	93	in Register window	181
SFR Setup window	183	using as assembler symbols	93
SFR/Uncached (Edit Memory Access option)	196	stack usage, computing	164
shifting, detecting bit loss or undefined behavior	303	Stack window	177
shortcut menu. <i>See</i> context menu		stack.mac	365
Show all images (Images window context menu)	68	Stall processor on FIFO full	
Show All (SFR Setup window context menu)	185	(ETM Trace Settings option)	211
Show Arguments (Call Stack window context menu)	83	standard C, sizeof operator in C-SPY	94
Show Custom SFRs only (SFR Setup		Start address (Fill option)	173
window context menu)	185	Start address (Memory Save option)	171
Show Cycles (Interrupt Log window context menu) .	361, 363	static analysis	
Show Cycles (Power Log window context menu)	289	documentation for	28
Show Factory SFRs only (SFR Setup		Statics window	109
window context menu)	185	Status (Cores window)	340
Show Numbers (Timeline window context menu)	232	Step Into (Debug menu)	63
Show Numerical Value (Timeline window context menu)	232	Step Into, description	73
Show offsets (Stack window context menu)	179	Step Out (Debug menu)	63
Show only (Image window context menu)	68	Step Out, description.	74
Show Time (Interrupt Log window context menu) .	360, 363	Step Over (Debug menu)	63
Show Time (Power Log window context menu)	289	Step Over, description.	73
Show timestamp (ETM Trace Settings option)	211	step points, definition of	72

--stlink_reset_strategy (C-SPY command line option) . . . 485
 Stop address (Memory Save option) 171
 Stop Debugging (Debug menu) 63
 __strFind (C-SPY system macro) 425
 ST-LINK communication problem 534
 ST-LINK (C-SPY driver), menu 553
 __subString (C-SPY system macro) 426
 Suppress download (debugger option) 500
 SWD interface, information in Trace window 203
 SWD (Interface setting) 518, 529–530
 Switch off after debugging (Target power supply setting) . 516
 switch, detect unhandled cases 304
 SWO clock (SWO Configuration option) 216
 SWO communication channel
 enabling 460, 518, 529–530, 536
 for timestamps in trace 212
 SWO Configuration dialog box 214
 SWO Configuration (I-jet/JTAGjet menu) 547
 SWO Configuration (J-Link menu) 550
 SWO Configuration (ST-LINK menu) 553
 SWO on the TraceD0 pin (I-jet/JTAGjet Trace option) . . . 522
 SWO prescaler (I-jet/JTAGjet Trace option) 522
 SWO prescaler (SWO Configuration option (I-jet)) 217
 SWO protocol (I-jet/JTAGjet option) 519
 SWO protocol (I-jet/JTAGjet Trace option) 522
 SWO trace 201
 SWO Trace Save (J-Link menu) 550
 SWO Trace Save (ST-LINK menu) 553
 SWO Trace Settings dialog box 212
 SWO Trace Settings dialog box (I-jet) 212
 SWO Trace Window Settings (I-jet/JTAGjet menu) 547
 SWO Trace Window Settings (J-Link menu) 550
 SWO Trace Window Settings (ST-LINK menu) 553
 SWO Trace (I-jet/JTAGjet menu) 547
 SWO Trace (J-Link menu) 551
 SWO Trace (ST-LINK menu) 553
 Symbolic Memory window 174
 Symbols window 114
 symbols, using in C-SPY expressions 92

System (Reset setting) 508, 514

T

TAP number (JTAG scan chain setting) 529
 Target number (Explicit probe configuration setting) 511, 518
 Target power (I-jet/JTAGjet option) 516
 target system, definition of 37
 __targetDebuggerVersion (C-SPY system macro) 426
 TCP/IP address or hostname (GDB Server option) 512
 TCP/IP Macraigor option) 530
 TCP/IP (Angel option) 506
 TCP/IP (Communication setting) 528
 Terminal IO Log Files (Terminal IO Log Files option) . . . 86
 Terminal I/O Log Files dialog box 85
 Terminal I/O window 77, 84
 terminology 29
 Text search (Find in Trace option) 251
 Third-Party Driver (debugger options) 539
 TI emulation package installation path
 (TI XDS option) 537
 TI Stellaris (C-SPY driver), menu 554
 TI XDS (C-SPY driver), menu 554
 Time Axis Unit (Timeline window context menu) 233
 time interval, in Timeline window 261
 Time Stamps (Force setting) 212
 Timeline window 224
 power graph 290
 Timeline (CMSIS-DAP menu) 545
 Timeline (I-jet/JTAGjet menu) 548
 Timeline (J-Link menu) 551
 Timeline (Simulator menu) 543
 Timeline (ST-LINK menu) 553
 --timeout (C-SPY command line option) 486
 timer interrupt, example 348
 timestamps in SWO trace 212
 Timestamps (SWO Configuration option) 217
 To Log File (ITM Stimulus Ports setting) 217

To Terminal I/O window (ITM Stimulus Ports setting)	217
Toggle Breakpoint (Code) (Call Stack window context menu)	83
Toggle Breakpoint (Code) (Disassembly window context menu)	81
Toggle Breakpoint (Log) (Call Stack window context menu)	83
Toggle Breakpoint (Log) (Disassembly window context menu)	81
Toggle Breakpoint (Trace Start) (Call Stack window context menu)	83
Toggle Breakpoint (Trace Start) (Disassembly window context menu)	81
Toggle Breakpoint (Trace Stop) (Call Stack window context menu)	83
Toggle Breakpoint (Trace Stop) (Disassembly window context menu)	81
Toggle source (Trace toolbar)	218
__toLower (C-SPY system macro)	427
tools icon, in this guide	30
__toString (C-SPY system macro)	427
__toUpper (C-SPY system macro)	428
trace	199
Trace buffer size (Trace Settings option)	209, 211
Trace Expressions window	250
Trace Filter breakpoints dialog box (J-Link)	247
Trace port mode (Trace Settings option)	208, 210
Trace port width (Trace Settings option)	208, 210
Trace Save dialog box	253
Trace Save (RDI menu)	552
Trace Settings (RDI menu)	552
trace start and stop breakpoints, overview	126
Trace Start breakpoints dialog box	235, 237, 239, 241–242
Trace Stop breakpoints dialog box	236
Trace window	218
trace (calls), profiling source	256, 264
trace (flat), profiling source	256, 264
Trace (Simulator menu)	542
trace, in Timeline window	224, 290

trademarks	2
Trigger at (Trace Start option)	237, 239, 241–242
Trigger at (Trace Stop option)	245
Trigger range (Data breakpoints option)	148
Trigger range (Data Log breakpoints option)	152
Trigger range (Trace Start option)	238, 240, 243, 246, 248
Trigger (Forced Interrupt window context menu)	355
typographic conventions	29

U

UART (SWO protocol setting)	522
Unavailable, C-SPY message	95
unhandled cases in switch statements, detecting	304
__unloadImage(C-SPY system macro)	428
USB (Communication setting)	528
Use command line options (debugger option)	503
Use Extra Images (debugger option)	501
Use flash loader (debugger option)	501
Use manual ranges (Memory Access Setup option)	189
Use ranges based on (Memory Access Setup option)	188
Use tab-separated format (Trace Save option)	253
Used ranges (Memory Configuration option)	193
user application, definition of	37
User (Mode setting)	144
using checked variant	295

V

Value (Address setting)	143
Value (Data setting)	144
Value (Fill option)	173
Value (Match data setting)	149, 238, 240, 244, 246, 249
variables	
effects of optimizations	95
information, limitation on	95
using in C-SPY expressions	93
variance (interrupt property), definition of	345
Variance % (Edit Interrupt option)	354

Vector Catch dialog box	156
Vector Catch (CMSIS-DAP menu)	545
Vector Catch (I-jet/JTAGjet menu)	548
Vector Catch (J-Link menu)	549
Vector Catch (Macraigor JTAG menu)	552
Verify download (debugger option)	500
version number	
of this guide	2
Viewing Range dialog box	233
Viewing Range (Timeline window context menu)	232
visualSTATE, C-SPY plugin module for	38

W

waiting for device, power consumption during	277
Wanted (SWO clock setting)	216
warnings icon, in this guide	30
Watch window	104
using	91
Watchpoints (J-Link menu)	549
Watchpoints (Macraigor JTAG menu)	552
web sites, recommended	29
while (macro statement)	375
windows, specific to C-SPY	65
Word (Data setting)	143
Write (Access Type setting)	143
__writeAPReg (C-SPY system macro)	429
__writeDPRReg (C-SPY system macro)	429
__writeFile (C-SPY system macro)	430
__writeFileByte (C-SPY system macro)	430
__writeMemoryByte (C-SPY system macro)	431
__writeMemory8 (C-SPY system macro)	431
__writeMemory16 (C-SPY system macro)	431
__writeMemory32 (C-SPY system macro)	432

X

--xds_board_file (C-SPY command line option)	486
--xds_reset_strategy (C-SPY command line option)	487

--xds_rootdir (C-SPY command line option)	487
---	-----

Z

zone	
in C-SPY	163
part of an absolute address	157
Zone (Edit SFR option)	187
Zoom (Timeline window context menu)	231

Symbols

__as_get_base (operator)	329
__as_get_bound (operator)	329
__as_make_bounds (operator)	330
__cancelAllInterrupts (C-SPY system macro)	383
__cancelInterrupt (C-SPY system macro)	384
__clearBreak (C-SPY system macro)	384
__closeFile (C-SPY system macro)	384
__delay (C-SPY system macro)	385
__disableInterrupts (C-SPY system macro)	385
__driverType (C-SPY system macro)	385
__emulatorSpeed (C-SPY system macro)	386
__emulatorStatusCheckOnRead (C-SPY system macro)	387
__enableInterrupts (C-SPY system macro)	388
__evaluate (C-SPY system macro)	388
__fillMemory8 (C-SPY system macro)	389
__fillMemory16 (C-SPY system macro)	389
__fillMemory32 (C-SPY system macro)	390
__fmessage (C-SPY macro statement)	375
__gdbserver_exec_command (C-SPY system macro)	391
__getSelectedCore (C-SPY system macro)	392
__getTracePortSize (C-SPY system macro)	392
__hasDAPRegs (C-SPY system macro)	393
__hwJetResetWithStrategy (C-SPY system macro)	393
__hwReset (C-SPY system macro)	394
__hwResetRunToBp (C-SPY system macro)	394
__hwResetWithStrategy (C-SPY system macro)	396
__hwRunToBreakpoint (C-SPY system macro)	396

__isBatchMode (C-SPY system macro)	397	__smessage (C-SPY macro statement)	375
__jlinkExecCommand (C-SPY system macro)	398	__sourcePosition (C-SPY system macro)	425
__jtagCommand (C-SPY system macro)	398	__strFind (C-SPY system macro)	425
__jtagCP15IsPresent (C-SPY system macro)	399	__subString (C-SPY system macro)	426
__jtagCP15ReadReg (C-SPY system macro)	399	__targetDebuggerVersion (C-SPY system macro)	426
__jtagCP15WriteReg (C-SPY system macro)	399	__toLower (C-SPY system macro)	427
__jtagData (C-SPY system macro)	400	__toString (C-SPY system macro)	427
__jtagRawRead (C-SPY system macro)	400	__toUpper (C-SPY system macro)	428
__jtagRawSync (C-SPY system macro)	401	__unloadImage (C-SPY system macro)	428
__jtagRawWrite (C-SPY system macro)	402	__writeAPReg (C-SPY system macro)	429
__jtagResetTRST (C-SPY system macro)	403	__writeDPRReg (C-SPY system macro)	429
__loadImage (C-SPY system macro)	403	__writeFile (C-SPY system macro)	430
__memoryRestore (C-SPY system macro)	404	__writeFileByte (C-SPY system macro)	430
__memorySave (C-SPY system macro)	405	__writeMemoryByte (C-SPY system macro)	431
__message (C-SPY macro statement)	375	__writeMemory8 (C-SPY system macro)	431
__messageBoxYesCancel (C-SPY system macro)	406	__writeMemory16 (C-SPY system macro)	431
__messageBoxYesNo (C-SPY system macro)	406	__writeMemory32 (C-SPY system macro)	432
__openFile (C-SPY system macro)	407	-f (cspybat option)	465
__orderInterrupt (C-SPY system macro)	408	-p (C-SPY command line option)	481
__popSimulatorInterruptExecutingStack (C-SPY system macro)	409	--backend (C-SPY command line option)	449
__readAPReg (C-SPY system macro)	409	--BE32 (C-SPY command line option)	442
__readDPRReg (C-SPY system macro)	410	--BE8 (C-SPY command line option)	442
__readFile (C-SPY system macro)	411	--bounds_table_size (linker option)	324
__readFileByte (C-SPY system macro)	411	--code_coverage_file (C-SPY command line option)	449
__readMemoryByte (C-SPY system macro)	412	--cpu (C-SPY command line option)	442
__readMemory8 (C-SPY system macro)	412	--cycles (C-SPY command line option)	450
__readMemory16 (C-SPY system macro)	412	--debugfile (cspybat option)	450
__readMemory32 (C-SPY system macro)	413	--debug_heap (linker option)	325
__registerMacroFile (C-SPY system macro)	413	--device (C-SPY command line option)	451
__resetFile (C-SPY system macro)	414	--disable_interrupts (C-SPY command line option)	451
__restoreSoftwareBreakpoints (C-SPY system macro)	414	--download_only (C-SPY command line option)	451
__selectCore (C-SPY system macro)	415	--drv_attach_to_program (C-SPY command line option)	443
__setCodeBreak (C-SPY system macro)	415	--drv_catch_exceptions (C-SPY command line option)	452
__setDataBreak (C-SPY system macro)	416	--drv_communication (C-SPY command line option)	453
__setDataLogBreak (C-SPY system macro)	418	--drv_communication_log (C-SPY command line option)	459
__setLogBreak (C-SPY system macro)	419	--drv_default_breakpoint (C-SPY command line option)	459
__setSimBreak (C-SPY system macro)	420	--drv_interface (C-SPY command line option)	460
__setTraceStartBreak (C-SPY system macro)	421	--drv_interface_speed (C-SPY command line option)	461
__setTraceStopBreak (C-SPY system macro)	423		

--drv_reset_to_cpu_start (C-SPY command line option) . 462
 --drv_restore_breakpoints
 (C-SPY command line option) 462
 --drv_suppress_download
 (C-SPY command line option) 443
 --drv_swo_clock_setup
 (C-SPY command line option) 463
 --drv_vector_table_base (C-SPY command line option). . 464
 --drv_verify_download (C-SPY command line option) . . 443
 --endian (C-SPY command line option) 443
 --flash_loader (C-SPY command line option). 465
 --fpu (C-SPY command line option). 443
 --gdbserv_exec_command
 (C-SPY command line option) 466
 --generate_entries_without_bounds (compiler option) . . 325
 --ignore_uninstrumented_pointers (compiler option) . . . 326
 --ignore_uninstrumented_pointers (linker option). 326
 --jet_board_cfg (C-SPY command line option) 466
 --jet_board_did (C-SPY command line option) 466
 --jet_cpu_clock (C-SPY command line option) 467
 --jet_ir_length (C-SPY command line option) 468
 --jet_power_from_probe (C-SPY command line option) . 468
 --jet_probe (C-SPY command line option) 469
 --jet_script_file (C-SPY command line option) 469
 --jet_standard_reset (C-SPY command line option) 470
 --jet_startup_connection_timeout (C-SPY command line option) 471
 --jet_swo_on_d0 (C-SPY command line option) 471
 --jet_swo_prescaler (C-SPY command line option) 472
 --jet_swo_protocol (C-SPY command line option). 472
 --jet_tap_position (C-SPY command line option). 473
 --jlink_dcc_timeout (C-SPY command line option) 473
 --jlink_device_select (C-SPY command line option) . . . 474
 --jlink_exec_command (C-SPY command line option) . 474
 --jlink_initial_speed (C-SPY command line option). . . . 474
 --jlink_ir_length (C-SPY command line option). 475
 --jlink_reset_strategy (C-SPY command line option) . . . 475
 --jlink_script_file (C-SPY command line option). 476
 --jlink_trace_source (C-SPY command line option) 476
 --macro (C-SPY command line option) 477

--macro_param (C-SPY command line option). 477
 --mac_handler_address (C-SPY command line option) . . 478
 --mac_jtag_device (C-SPY command line option) 478
 --mac_multiple_targets (C-SPY command line option) . . 479
 --mac_reset_pulls_reset (C-SPY command line option) . . 479
 --mac_set_temp_reg_buffer
 (C-SPY command line option) 480
 --mac_xscale_ir7 (C-SPY command line option) 480
 --mapu (C-SPY command line option) 480
 --plugin (C-SPY command line option) 481
 --proc_stack_xxx (C-SPY command line option) 482
 --rdi_allow_hardware_reset
 (C-SPY command line option) 482
 --rdi_driver_dll (C-SPY command line option). 483
 --rdi_heartbeat (C-SPY command line option) 444
 --rdi_step_max_one (C-SPY command line option) 483
 --reset_style (C-SPY command line option) 483
 --rtc_enable (cspsybat option)). 330
 --rtc_output (cspsybat option)). 331
 --rtc_raw_to_txt (cspsybat option)) 331
 --rtc_rules (cspsybat option)) 332
 --runtime_checking (compiler option) 326
 --semihosting (C-SPY command line option) 484
 --silent (C-SPY command line option) 485
 --stlink_reset_strategy (C-SPY command line option) . . 485
 --timeout (C-SPY command line option) 486
 --xds_board_file (C-SPY command line option). 486
 --xds_reset_strategy (C-SPY command line option). . . . 487
 --xds_rootdir (C-SPY command line option) 487

Numerics

1x Units (Symbolic Memory window context menu) 176
 8x Units (Memory window context menu) 169