

C-SPY Debugging Guide

for Arm Limited's Arm© cores

May 2025
UCSARM-28

Copyright © 1999-2025 IAR Systems AB

Copyright notice

This document contains IAR Systems AB (hereinafter “IAR”) proprietary information and may, in no part, be reproduced without the prior written consent of IAR. The software described in this document is furnished under a license and may only be installed, used and/or copied in accordance with the terms and conditions of such license.

Export control

The software described herein and thereto related technical information may be subject to Swedish, EU and/or US export control regulations. As such, the aforementioned technical information contained herein may not be disclosed, exported or re-exported contrary to such export control regulations, nor may it be shared with individuals or entities subject trade restrictions or other international sanctions.

Disclaimer

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR. While the information contained herein is assumed to be accurate, it is provided as-is and IAR 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, IAR Embedded Workbench, Embedded Trust, C-Trust, IAR Connect, C-SPY, C-RUN, C-STAT, IAR Visual State, IAR KickStart Kit, I-jet, I-jet Trace, I-scope, IAR Academy, IAR, and the logotype of IAR are trademarks or registered trademarks owned by IAR Systems AB.

All other third party brands and product names referred to herein are trademarks or registered trademarks of their respective owners.

Table of Contents

Preface **16**
Who should read this guide 16
Required knowledge 16
How to use this guide 16
What this guide contains 17
Part 1. Basic debugging 17
Part 2. Analyzing your application 17
Part 3. Advanced debugging 18
Part 4. Additional reference information 18
Other documentation 18
User and reference guides 18
The online help system 18
Web sites 19
Document conventions 19
Typographic conventions 19
Naming conventions 20
Part 1. Basic debugging **21**
The IAR C-SPY Debugger **22**
Introduction to C-SPY 22
An integrated environment 22
General C-SPY debugger features 23
RTOS awareness 24
Debugger concepts 24
C-SPY and target systems 24
The debugger 25
The target system 25
The application 25
C-SPY debugger systems 25
The ROM-monitor program 26
Third-party debuggers 26
C-SPY plugin modules 26
Execution modes 26
C-SPY drivers overview 26
Differences between the C-SPY drivers 27
The IAR C-SPY Simulator 28
Two simulator drivers 28
Supported features 28
The C-SPY hardware debugger drivers 29
Communication overview 29
Hardware installation 30
USB driver installation 31
Getting started using C-SPY **33**
Setting up C-SPY 33
Setting up for debugging 33
Executing from reset 34
Using a setup macro file 34
Selecting a device description file 35
Loading plugin modules 35
Starting C-SPY 35
Starting a debug session 35
Loading executable files built outside of the IDE 35
Starting a debug session with source files missing 36

Loading multiple debug images ... 36
Editing in C-SPY windows ... 37
Adapting for target hardware ... 38
Modifying a device description file ... 38
Initializing target hardware before C-SPY starts ... 38
Remapping memory ... 38
Using predefined C-SPY macros for device support ... 39
Debug authentication ... 39
An overview of the debugger startup ... 40
Debugging code in flash ... 40
Debugging code in RAM ... 41
Reference information on starting C-SPY ... 41
C-SPY Debugger main window ... 41
Images window ... 46
Get Alternative File dialog box ... 47
Executing your application ... 49
Introduction to application execution ... 49
Briefly about application execution ... 49
Source and disassembly mode debugging ... 49
Single stepping ... 50
Troubleshooting slow stepping speed ... 51
Running the application ... 52
Highlighting ... 52
Viewing the call stack ... 53
Terminal input and output ... 54
Debug logging ... 54
Reference information on application execution ... 54
Disassembly window ... 55
Call Stack window ... 59
Terminal I/O window ... 61
Terminal I/O Log File dialog box ... 62
Debug Log window ... 63
Fault exception viewer window ... 64
Report Assert dialog box ... 65
Autostep settings dialog box ... 66
Variables and expressions ... 67
Introduction to working with variables and expressions ... 67
Briefly about working with variables and expressions ... 67
C-SPY expressions ... 68
Limitations on variable information ... 70
Working with variables and expressions ... 70
Using the windows related to variables and expressions ... 70
Viewing assembler variables ... 71
Reference information on working with variables and expressions ... 71
Auto window ... 71
Locals window ... 74
Watch window ... 76
Live Watch window ... 78
Statics window ... 81
Quick Watch window ... 84
Symbols window ... 87
Resolve Symbol Ambiguity dialog box ... 89
Breakpoints ... 90
Introduction to setting and using breakpoints ... 90
Reasons for using breakpoints ... 90

Briefly about setting breakpoints ... 91
Breakpoint types ... 91
Breakpoint icons ... 93
Breakpoints in the C-SPY simulator ... 93
Breakpoints in the C-SPY hardware debugger drivers ... 93
Breakpoint consumers ... 93
Breakpoint options ... 94
Setting breakpoints ... 94
Various ways to set a breakpoint ... 94
Toggling a simple code breakpoint ... 95
Setting breakpoints using the dialog box ... 95
Setting a data breakpoint in the Memory window ... 96
Setting breakpoints using system macros ... 96
Setting a breakpoint on an exception vector ... 97
Setting breakpoints in __ramfunc declared functions ... 98
Useful breakpoint hints ... 98
Reference information on breakpoints ... 99
Breakpoints window ... 99
Breakpoint Usage window ... 101
Code breakpoints dialog box ... 102
JTAG Watchpoints dialog box ... 103
Log breakpoints dialog box ... 106
Data breakpoints dialog box ... 107
Data Log breakpoints dialog box ... 110
Data Log breakpoints dialog box (C-SPY hardware drivers) ... 111
Breakpoints options ... 112
Immediate breakpoints dialog box ... 114
Vector Catch dialog box ... 115
Flash breakpoints dialog box ... 116
Enter Location dialog box ... 117
Resolve Source Ambiguity dialog box ... 118
Memory and registers ... 120
Introduction to monitoring memory and registers ... 120
Briefly about monitoring memory and registers ... 120
C-SPY memory zones ... 121
Memory configuration for the C-SPY simulator ... 122
Memory configuration for C-SPY hardware debugger drivers ... 122
Monitoring memory and registers ... 123
Defining application-specific register groups ... 123
Monitoring stack usage ... 123
Reference information on memory and registers ... 126
Memory window ... 126
Memory Save dialog box ... 129
Memory Restore dialog box ... 130
Fill dialog box ... 131
Symbolic Memory window ... 132
Stack window ... 135
Registers window ... 138
Register User Groups Setup window ... 141
SFR Setup window ... 142
Edit SFR dialog box ... 145
Memory Configuration dialog box for the C-SPY simulator ... 146
Edit Memory Range dialog box for the C-SPY simulator ... 148
Memory Configuration dialog box for C-SPY hardware debugger drivers ... 149
Edit Memory Range dialog box for C-SPY hardware debugger drivers ... 151

Part 2. Analyzing your application 154
Trace 155
Introduction to using trace 155
Reasons for using trace 155
Briefly about trace 156
Requirements for using trace 157
Collecting and using trace data 158
Getting started with ETM trace 158
Getting started with SWO trace 159
Getting started with MTB trace 159
Setting up concurrent use of ETM and SWO 160
Trace data collection using breakpoints 160
Searching in trace data 160
Browsing through trace data 161
Reference information on trace 161
ETM Trace Settings dialog box (I-jet) 161
ETM Trace Settings dialog box (J-Link/J-Trace) 163
SWO Trace Window Settings dialog box 165
SWO Configuration dialog box 167
Trace window 170
Function Trace window 178
Trace Start Trigger breakpoint dialog box 179
Trace Stop Trigger breakpoint dialog box 180
Trace Start breakpoints dialog box (I-jet and CMSIS-DAP) 181
Trace Stop breakpoints dialog box (I-jet and CMSIS-DAP) 183
Trace Filter breakpoints dialog box (I-jet) 184
Trace Start breakpoints dialog box (J-Link/J-Trace) 185
Trace Stop breakpoints dialog box (J-Link/J-Trace) 187
Trace Filter breakpoints dialog box (J-Link/J-Trace) 189
Find in Trace dialog box 191
Find in Trace window 193
Trace Save dialog box 193
The application timeline 195
Introduction to analyzing your application's timeline 195
Briefly about analyzing the timeline 195
Requirements for timeline support 196
Analyzing your application's timeline 197
Displaying a graph in the Timeline window 197
Navigating in the graphs 197
Analyzing performance using the graph data 198
Getting started using data logging 198
Getting started using event logging 199
Reference information on application timeline 200
Data Log window 201
Data Log Summary window 203
Event Log window 205
Event Log Summary window 207
Timeline window—Call Stack graph 211
Timeline window—Data Log graph 215
Timeline window—Events graph 218
Viewing Range dialog box 221
Profiling 224
Introduction to the profiler 224
Reasons for using the profiler 224
Briefly about the profiler 224

Requirements for using the profiler ... 225
Using the profiler ... 226
Getting started using the profiler on function level ... 226
Analyzing the profiling data ... 226
Getting started using the profiler on instruction level ... 228
Selecting a time interval for profiling information ... 228
Reference information on the profiler ... 229
Function Profiler window ... 229
Code coverage ... 233
Introduction to code coverage ... 233
Reasons for using code coverage ... 233
Briefly about code coverage ... 233
Requirements and restrictions for using code coverage ... 233
Reference information on code coverage ... 233
Code Coverage window ... 234
Performance monitoring ... 238
Introduction to performance monitoring ... 238
Briefly about performance monitoring ... 238
Requirements and restrictions for using performance monitoring ... 238
Event types ... 238
Detecting counter overflow ... 239
Setting up performance monitoring ... 239
Reference information on performance monitoring ... 240
Performance Monitoring window ... 240
Power debugging ... 243
Introduction to power debugging ... 243
Reasons for using power debugging ... 243
Briefly about power debugging ... 243
Requirements and restrictions for power debugging ... 244
Optimizing your source code for power consumption ... 244
Waiting for device status ... 245
Software delays ... 245
DMA versus polled I/O ... 245
Low-power mode diagnostics ... 245
CPU frequency ... 245
Detecting mistakenly unattended peripherals ... 246
Peripheral units in an event-driven system ... 246
Finding conflicting hardware setups ... 247
Analog interference ... 247
Debugging in the power domain ... 247
Displaying a power profile and analyzing the result ... 248
Detecting unexpected power usage during application execution ... 248
Changing the graph resolution ... 249
Reference information on power debugging ... 249
Power Log Setup window ... 249
Power Log window ... 251
Timeline window—Power graph ... 254
State Log window ... 255
State Log Summary window ... 256
Timeline window—State Log graph ... 259
C-RUN runtime error checking ... 263
Introduction to runtime error checking ... 264
Runtime error checking ... 264
Runtime error checking using C-RUN ... 264
The checked heap provided by the library ... 265

Using C-RUN in the IAR Embedded Workbench IDE ... 265
Using C-RUN in non-interactive mode ... 265
Requirements for runtime error checking ... 266
Using C-RUN ... 266
Getting started using C-RUN runtime error checking ... 266
Creating rules for messages ... 267
Detecting various runtime errors ... 267
Detecting implicit or explicit integer conversion ... 267
Detecting signed or unsigned overflow ... 269
Detecting bit loss or undefined behavior when shifting ... 270
Detecting division by zero ... 271
Detecting unhandled cases in switch statements ... 272
Detecting accesses outside the bounds of arrays and other objects ... 273
Detecting heap usage error ... 277
Detecting heap memory leaks ... 279
Detecting heap integrity violations ... 280
Reference information on runtime error checking ... 283
C-RUN Runtime Checking options ... 283
C-RUN Messages window ... 285
C-RUN Messages Rules window ... 287
Compiler and linker reference for C-RUN ... 288
--bounds_table_size ... 288
--debug_heap ... 289
--generate_entries_without_bounds ... 289
--ignore_uninstrumented_pointers ... 289
--ignore_uninstrumented_pointers ... 290
--runtime_checking ... 290
#pragma default_no_bounds ... 291
#pragma define_with_bounds ... 291
#pragma define_without_bounds ... 291
#pragma disable_check ... 292
#pragma generate_entry_without_bounds ... 292
#pragma no_arith_checks ... 292
#pragma no_bounds ... 293
__as_get_base ... 293
__as_get_bound ... 293
__as_make_bounds ... 294
cspybat options for C-RUN ... 294
--rtc_enable ... 294
--rtc_output ... 294
--rtc_raw_to_txt ... 295
--rtc_rules ... 295
Part 3. Advanced debugging ... 297
Multicore debugging ... 298
Introduction to multicore debugging ... 298
Briefly about multicore debugging ... 298
Symmetric multicore debugging ... 298
Asymmetric multicore debugging ... 299
Requirements and restrictions for multicore debugging ... 299
Debugging multiple cores ... 299
Setting up for symmetric multicore debugging ... 299
Setting up for asymmetric multicore debugging ... 300
Starting and stopping a multicore debug session ... 301
Reference information on multicore debugging ... 302
Cores window ... 302

Multicore toolbar 304
The multicore session file 304
Interrupts 306
Introduction to interrupts 306
Briefly about the interrupt simulation system 306
Interrupt characteristics 307
Interrupt simulation states 308
C-SPY system macros for interrupt simulation 308
Target-adapting the interrupt simulation system 309
Briefly about interrupt logging 309
Using the interrupt system 309
Simulating a simple interrupt 310
Simulating an interrupt in a multi-task system 311
Getting started using interrupt logging 311
Reference information on interrupts 312
Interrupt Configuration window 312
Available Interrupts window 314
Interrupt Status window 316
Interrupt Log window 318
Interrupt Log Summary window 321
Timeline window—Interrupt Log graph 324
C-SPY macros 328
Introduction to C-SPY macros 330
Reasons for using C-SPY macros 330
Briefly about using C-SPY macros 330
Briefly about setup macro functions and files 331
Briefly about the macro language 331
Using C-SPY macros 332
Registering C-SPY macros—an overview 332
Executing C-SPY macros—an overview 332
Registering and executing using setup macros and setup files 332
Executing macros using Quick Watch 333
Executing a macro by connecting it to a breakpoint 334
Aborting a C-SPY macro 334
Reference information on the macro language 335
Macro functions 335
Macro variables 335
Macro parameters 335
Macro strings 336
Macro statements 336
Formatted output 337
Reference information on reserved setup macro function names 338
execConfigureTraceETM 339
execConfigureTraceSWO 339
execUserAttach 339
execUserPreload 339
execUserExecutionStarted 340
execUserExecutionStopped 340
execUserFlashInit 340
execUserSetup 340
execUserFlashReset 341
execUserPreReset 341
execUserReset 341
execUserStatus 342
execUserExit 342

execUserFlashExit ... 342
execUserCoreConnect ... 343
Reference information on C-SPY system macros ... 343
Summary of C-SPY system macros ... 343
__abortLaunch ... 347
__cancelAllInterrupts ... 347
__cancelInterrupt ... 347
__clearBreak ... 348
__closeFile ... 348
__delay ... 349
__disableInterrupts ... 349
__driverType ... 349
__emulatorSpeed ... 350
__emulatorStatusCheckOnRead ... 351
__enableInterrupts ... 351
__evaluate ... 352
__expandVar ... 352
__fillMemory8 ... 353
__fillMemory16 ... 354
__fillMemory32 ... 354
__fillMemory64 ... 355
__gdbserver_exec_command ... 356
__getNumberOfCores ... 356
__getSelectedCore ... 356
__getTracePortSize ... 357
__hasDAPRegs ... 358
__hwJetResetWithStrategy ... 358
__hwReset ... 359
__hwResetRunToBp ... 359
__hwResetWithStrategy ... 360
__hwRunToBreakpoint ... 361
__isBatchMode ... 362
__isMacroSymbolDefined ... 362
__jlinkExecCommand ... 363
__jlinkExecMacro ... 363
__jtagCommand ... 364
__jtagCP15IsPresent ... 364
__jtagCP15ReadReg ... 365
__jtagCP15WriteReg ... 365
__jtagData ... 365
__jtagRawRead ... 366
__jtagRawSync ... 366
__jtagRawWrite ... 367
__jtagResetTRST ... 368
__loadImage ... 368
__memoryRestore ... 369
__memorySave ... 370
__messageBoxYesCancel ... 370
__messageBoxYesNo ... 371
__openFile ... 371
__orderInterrupt ... 372
__popSimulatorInterruptExecutingStack ... 373
__probeType ... 374
__readAPReg ... 374
__readDPReg ... 375

__readFile ... 375
__readFileByte ... 376
__readMemory8, __readMemoryByte ... 376
__readMemory16 ... 377
__readMemory32 ... 377
__readMemory64 ... 377
__registerMacroFile ... 378
__resetFile ... 378
__restoreSoftwareBreakpoints ... 379
__selectCore ... 379
__setCodeBreak ... 380
__setDataBreak ... 381
__setDataLogBreak ... 383
__setLogBreak ... 384
__setSimBreak ... 385
__setSystemMaxData ... 385
__setSystemTimeout ... 386
__setTraceStartBreak ... 387
__setTraceStopBreak ... 388
__sourcePosition ... 389
__strFind ... 390
__subString ... 390
__system1 ... 391
__system2 ... 392
__system3 ... 392
__targetDebuggerVersion ... 393
__toLower ... 393
__toString ... 394
__toUpper ... 395
__unloadImage ... 395
__wallTime_ms ... 396
__whichCore ... 396
__writeAPReg ... 397
__writeDPReg ... 397
__writeFile ... 398
__writeFileByte ... 398
__writeMemory8, __writeMemoryByte ... 399
__writeMemory16 ... 399
__writeMemory32 ... 399
__writeMemory64 ... 400
Graphical environment for macros ... 401
Macro Registration window ... 401
Debugger Macros window ... 402
Macro Quicklaunch window ... 404
The C-SPY command line utility—cspybat ... 407
Using C-SPY in batch mode ... 409
Starting cspybat ... 409
Output ... 409
Invocation syntax ... 409
Summary of C-SPY command line options ... 410
General cspybat options ... 410
Options available for all C-SPY drivers ... 411
Options available for the simulator driver ... 411
Options available for the C-SPY GDB Server driver ... 412
Options available for the C-SPY I-jet driver ... 412

Options available for the C-SPY CMSIS-DAP driver 413
Options available for the C-SPY J-Link/J-Trace driver 414
Options available for the C-SPY TI MSP-FET driver 414
Options available for the C-SPY TI Stellaris driver 415
Options available for the C-SPY TI XDS driver 415
Options available for the C-SPY ST-LINK driver 415
Options available for the C-SPY third-party drivers 416
Reference information on C-SPY command line options 416
--application_args 416
--attach_to_running_target 417
--backend 417
--board_file 417
--code_coverage_file 418
--cycles 418
--debug_auth_enforce 419
--debug_auth_settings 419
--debug_auth_type 420
--debug_file 420
--device 420
--device_macro 421
--disable_interrupts 421
--download_only 422
--drv_catch_exceptions 422
--drv_communication 423
--drv_debugger_cache 425
--drv_default_breakpoint 426
--drv_enforce_mem_config 426
--drv_exclude_from_verify 427
--drv_interface 427
--drv_interface_speed 428
--drv_mem_ap 429
--drv_reset_to_cpu_start 429
--drv_restore_breakpoints 430
--drv_swo_clock_setup 430
--drv_trace_settings 431
--drv_vector_table_base 432
-f 432
--flash_loader 433
--function_profiling 433
--gdbserv_exec_command 434
--jet_board_cfg 434
--jet_board_did 434
--jet_cpu_clock 435
--jet_disable_pmu 436
--jet_disable_pmu_dap 436
--jet_ir_length 436
--jet_power_from_probe 437
--jet_probe 437
--jet_script_file 437
--jet_standard_reset 438
--jet_startup_connection_timeout 439
--jet_swo_on_d0 439
--jet_swo_prescaler 440
--jet_swo_protocol 440
--jet_tap_position 441

--jlink_dcc_timeout ... 441
--jlink_device_select ... 441
--jlink_exec_command ... 442
--jlink_initial_speed ... 442
--jlink_ir_length ... 443
--jlink_reset_strategy ... 443
--jlink_script_file ... 444
--jlink_trace_source ... 444
--leave_target_running ... 445
--lmiftdi_reset_strategy ... 445
--macro ... 445
--macro_param ... 446
--mapu ... 446
--mspfet_erase_flash ... 447
--mspfet_interface_speed ... 447
--mspfet_reset_strategy ... 448
--mspfet_settlingtime ... 448
--mspfet_vccvoltage ... 448
--multicore_nr_of_cores ... 449
-p ... 449
--plugin ... 450
--proc_stack_*stack* ... 450
--reset_style ... 451
--sdm_debug_architecture ... 452
--sdm_library ... 452
--sdm_library_hint ... 453
--sdm_manifest ... 453
--semihosting ... 454
--silent ... 454
--sockets ... 455
--stlink_reset_strategy ... 455
--suppress_entrypoint_warning ... 456
--timeout ... 456
--xds_board_file ... 456
--xds_reset_strategy ... 457
--xds_rootdir ... 458
Flash loaders ... 459
Introduction to the flash loader ... 459
Using flash loaders ... 459
Setting up the flash loader(s) ... 459
The flash loading mechanism ... 459
Aborting a flash loader ... 460
Reference information on the flash loader ... 460
Flash Loader Overview dialog box ... 460
Flash Loader Configuration dialog box ... 462
Part 4. Additional reference information ... 464
Debugger options ... 465
Setting debugger options ... 465
Reference information on general debugger options ... 467
Setup ... 467
Download ... 468
Images ... 469
Multicore ... 470
Authentication ... 471
Extra Options ... 473

Plugins 474
Reference information on C-SPY hardware debugger driver options 475
CADI — Setup 475
CMSIS-DAP — Setup 476
CMSIS-DAP — Interface 479
E2 — Setup 480
GDB Server 481
G+LINK — Setup 482
I-jet — Setup 482
I-jet — Interface 485
I-jet — Trace 487
J-Link/J-Trace — Setup 490
J-Link/J-Trace — Connection 494
Nu-Link — Setup 495
PE micro — Setup 496
ST-LINK — Setup 497
ST-LINK — Communication 499
ST-LINK — Multicore 500
TI MSP-FET — Setup 500
TI MSP-FET — Download 502
TI Stellaris — Setup 502
TI XDS — Setup 503
TI XDS — Communication 505
Third-Party Driver options 507
Additional information on C-SPY drivers 509
Reference information on C-SPY driver menus 509
C-SPY driver 510
Simulator menu (IAR native) 510
Simulator menu (Imperas) 512
Reference information on the C-SPY simulator 512
Simulated Frequency dialog box 512
Reference information on the C-SPY hardware debugger drivers 513
CADI menu 513
CMSIS-DAP menu 513
E2 menu 515
GDB Server menu 516
G+LINK menu 517
I-jet menu 517
J-Link menu 520
Nu-Link menu 523
ST-Link menu 524
TI MSP-FET menu 526
TI Stellaris menu 527
TI XDS menu 528
Resolving problems 531
No contact with the target hardware 531

List of Tables

1. Typographic conventions used in IAR documentation 19
2. Naming conventions used in IAR documentation 20
3. Driver differences—Simulator, I-jet, J-Link/J-Trace, ST-LINK,TI XDS, etc 27
4. Driver differences—other drivers 28
5. Terminal I/O in real time 61
6. C-SPY assembler symbols expressions 69
7. Handling name conflicts between hardware registers and assembler labels 69

8. Live watch for the different devices 78
9. C-SPY macros for breakpoints 97
10. C-SPY macros for breakpoints 97
11. Support for timeline information 196
12. C-SPY driver profiling support 225
13. Project options for enabling the profiler 226
14. Timer interrupt settings 310
15. Examples of C-SPY macro variables 335
16. Summary of system macros 343
17. __cancelInterrupt return values 348
18. __disableInterrupts return values 349
19. __driverType return values 350
20. __emulatorSpeed return values 350
21. __enableInterrupts return values 351
22. __evaluate return values 352
23. __expandVar return values 352
24. __getTracePortSize return values 357
25. __hasDAPRegs return values 358
26. __hwJetResetWithStrategy return values 358
27. __hwReset return values 359
28. __hwResetRunToBp return values 360
29. __hwResetWithStrategy return values 360
30. __hwRunToBreakpoint return values 361
31. __isBatchMode return values 362
32. __jtagResetTRST return values 368
33. __loadImage return values 368
34. __messageBoxYesCancel return values 371
35. __messageBoxYesNo return values 371
36. __openFile return values 372
37. __probeType return values 374
38. __readAPReg return values 374
39. __readDPReg return values 375
40. __readFile return values 375
41. __setCodeBreak return values 380
42. __setDataBreak return values 382
43. __setDataLogBreak return values 383
44. __setLogBreak return values 384
45. __setSimBreak return values 385
46. __setTraceStartBreak return values 388
47. __setTraceStopBreak return values 389
48. __sourcePosition return values 389
49. __unloadImage return values 395
50. __writeAPReg return values 397
51. __writeDPReg return values 397
52. cspybat parameters 410
53. Options specific to the C-SPY drivers you are using 466

Preface

Contents

Who should read this guide 16
Required knowledge 16
How to use this guide 16
What this guide contains 17
Part 1. Basic debugging 17
Part 2. Analyzing your application 17
Part 3. Advanced debugging 18
Part 4. Additional reference information 18
Other documentation 18
User and reference guides 18
The online help system 18
Web sites 19
Document conventions 19
Typographic conventions 19
Naming conventions 20

Welcome to the *C-SPY Debugging Guide for Arm*, which helps you to 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 you are using (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

For more information about the other development tools incorporated in the IDE, refer to their respective documentation, see *Other documentation, page 18*.

HOW TO USE THIS GUIDE

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

- *Concepts*, which describes the topic and gives overviews of features related to the topic area. Any requirements or restrictions are also listed. Read this section to learn about the topic area.
- *Tasks*, which lists useful tasks related to the topic area. For many of the tasks, you can also find step-by-step descriptions. Read this section for information about required tasks as well as for information about how to perform certain tasks.

- *Reference information*, which gives reference information related to the topic area. Read this section for information about certain features or GUI components.

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

Finally, we recommend the *Glossary* in the *IDE Project Management and Building Guide for Arm* if you should encounter any unfamiliar terms in the IAR user documentation.

WHAT THIS GUIDE CONTAINS

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



Some of the screenshots in this guide are taken from a similar product and not from IAR Embedded Workbench for Arm.

Part 1. Basic debugging

- *The IAR C-SPY Debugger, page 22* 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, page 33* helps you get started using C-SPY, which includes setting up, starting, and adapting C-SPY for target hardware.
- *Executing your application, page 49* 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, page 67* 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, page 90* describes the breakpoint system and the various ways to set breakpoints.
- *Memory and registers, page 120* shows how you can examine memory and registers.

Part 2. Analyzing your application

- *Trace, page 155* describes how you can inspect the program flow up to a specific state using trace data.
- *The application timeline, page 195* describes the **Timeline** window, and how to use the information in it to analyze your application's behavior.
- *Profiling, page 224* 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, page 233* 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.
- *Performance monitoring, page 238* describes the **Performance Monitoring** window, and how to use this window to view event counters or CPU clock cycles through the Performance Monitoring Unit (PMU).
- *Power debugging, page 243* 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, page 263* describes how to use C-RUN for runtime error checking.

Part 3. Advanced debugging

- *Multicore debugging, page 298* describes how to debug a target with multiple cores.
- *Interrupts, page 306* contains detailed information about the C-SPY interrupt simulation system and how to configure the simulated interrupts to make them reflect the interrupts of your target hardware.
- *C-SPY macros, page 328* 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, page 407* describes how to use C-SPY in batch mode.
- *Flash loaders, page 459* describes the flash loader, what it is and how to use it.

Part 4. Additional reference information

- *Debugger options, page 465* describes the options you must set before you start the C-SPY debugger.
- *Additional information on C-SPY drivers, page 509* 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 help system in HTML format. You can access the documentation from the IAR Information Center or from the **Help** menu in the IAR Embedded Workbench IDE.

User and reference guides

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

- System requirements and information about how to install and register the IAR products are available in the *Installation and Licensing Quick Reference Guide* and the *Licensing Guide*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for Arm.*
- Using the IAR C-SPY® Debugger and C-RUN runtime error checking, is available in the *C-SPY Debugging Guide for Arm.*
- Programming for the IAR C/C++ Compiler for Arm and linking, is available in the *IAR C/C++ Development Guide for Arm.*
- Programming for the IAR Assembler for Arm is available in the *IAR Assembler User Guide for Arm.*
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT® Static Analysis Guide*.
- Using I-jet, refer to the *IAR Debug Probes User Guide for I-jet®, I-jet Trace, and I-scope*.
- Using J-Link and J-Trace, refer to the J-Link/J-Trace documentation available at *www.segger.com*.
- 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*.



Additional documentation might be available depending on your product installation.

The online help system

The context-sensitive online help contains information about:

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

- C-STAT

Web sites

Recommended web sites:

- The chip manufacturer's web site.
- The Arm Limited web site, **www.arm.com**, that contains information and news about the Arm cores.
- The IAR web site, **www.iar.com**, that holds application notes and other product information.
- The web site of the C standardization working group, **www.open-std.org/jtc1/sc22/wg14**.
- The web site of the C++ Standards Committee, **www.open-std.org/jtc1/sc22/wg21**.
- The C++ programming language web site, **isocpp.org**. This web site also has a list of recommended books about C++ programming.
- The C and C++ reference web site, **en.cppreference.com**.

DOCUMENT CONVENTIONS

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

When referring to a directory in your product installation, for example `arm\doc`, the full path to the location is assumed, for example `c:\iar\ewarm-2.1\arm\doc`.

Typographic conventions

The IAR documentation set uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
parameter	A placeholder for an actual value used as a parameter, for example *filename*`.h` where *filename* represents the name of the file.
[`option`**]**	An optional part of a linker or stack usage control directive, where **[** and **]** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
{`option`**}**	A mandatory part of a linker or stack usage control directive, where **{** and **}** are not part of the actual directive, but any `[`, `]`, `{`, or `}` are part of the directive syntax.
`[option]`	An optional part of a command line option, pragma directive, or library filename.
`[a|b|c]`	An optional part of a command line option, pragma directive, or library filename with alternatives.
`{a|b|c}`	A mandatory part of a command line option, pragma directive, or library filename with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• 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.

Style	Used for
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1. Typographic conventions used in IAR documentation

Naming conventions

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

Brand name	Generic term
IAR Embedded Workbench® for 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 IAR documentation

In 32-bit mode refers to using IAR Embedded Workbench for Arm configured for the instruction sets T32/T and A32.

In 64-bit mode refers to using IAR Embedded Workbench for Arm configured for the instruction set A64.

For more information, see *Execution modes, page 26*.

Part 1. Basic debugging

Part 1. Basic debugging

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

- *The IAR C-SPY Debugger, page 22*
- *Getting started using C-SPY, page 33*
- *Executing your application, page 49*
- *Variables and expressions, page 67*
- *Breakpoints, page 90*
- *Memory and registers, page 120*

The IAR C-SPY Debugger

Contents

Introduction to C-SPY 22
An integrated environment 22
General C-SPY debugger features 23
RTOS awareness 24
Debugger concepts 24
C-SPY and target systems 24
The debugger 25
The target system 25
The application 25
C-SPY debugger systems 25
The ROM-monitor program 26
Third-party debuggers 26
C-SPY plugin modules 26
Execution modes 26
C-SPY drivers overview 26
Differences between the C-SPY drivers 27
The IAR C-SPY Simulator 28
Two simulator drivers 28
Supported features 28
The C-SPY hardware debugger drivers 29
Communication overview 29
Hardware installation 30
USB driver installation 31

INTRODUCTION TO C-SPY

An integrated environment

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

- *Editing while debugging*

  During a debug session, you can make corrections directly in the same source code window that is used for controlling the debugging. Changes will be included in the next project rebuild.
- *Setting breakpoints at any point during the development cycle*

  You can inspect and modify breakpoint definitions also when the debugger is not running, and breakpoint definitions flow with the text as you edit. Your debug settings, such as watch properties, window layouts, and register groups will be preserved between your debug sessions.

  Note: Setting breakpoints when the debugger is running is not supported by the C-SPY Simulator for all cores or devices.

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

General C-SPY debugger features

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

C-SPY offers these general features:

- *Source and disassembly level debugging*

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

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

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

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

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

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

  C-SPY includes a powerful internal macro system, to allow you to define complex sets of actions to be performed. C-SPY macros can be used on their own or in conjunction with complex breakpoints and—for some cores or devices—the interrupt simulation system to perform a wide variety of tasks.

Additional general C-SPY debugger features

This list shows some additional features:

- Threaded execution keeps the IDE responsive while running the target application
- Automatic stepping
- The source browser provides easy navigation to functions, types, and variables
- Extensive type recognition of variables
- Configurable registers (CPU and peripherals) and memory windows
- Graphical stack view with overflow detection
- Support for code coverage (not supported by the C-SPY Simulator for all cores or devices)
- Support for performance monitoring of event counters and CPU clock cycles through the Performance Monitoring Unit (PMU)
- 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
- Azure RTOS ThreadX
- CMX-RTX
- CMX-Tiny+
- eForce mC3/Compact
- eSysTech X realtime kernel
- FreeRTOS, OpenRTOS, and SafeRTOS
- Freescale MQX
- Micrium uC/OS-II
- Micrium uC/OS-III
- Micro Digital SMX
- MISPO NORTi
- OSEK Run Time Interface (ORTI)
- RTXC Quadros
- Segger embOS
- unicoi Fusion

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

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

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

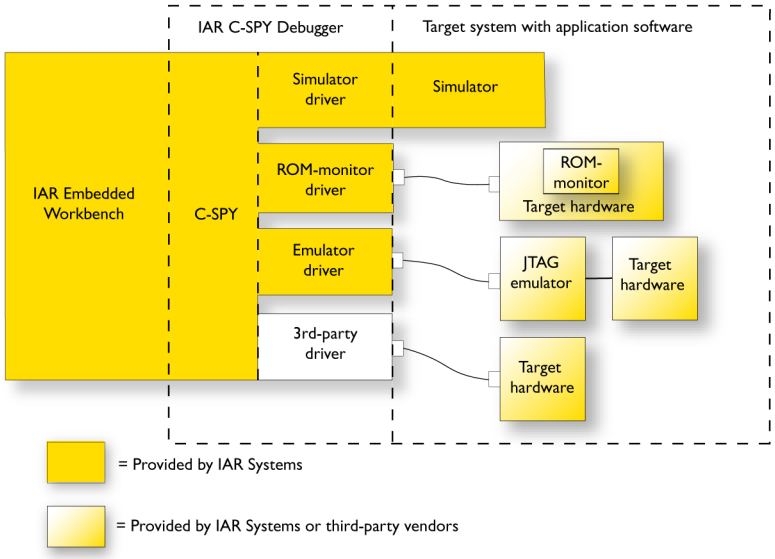
DEBUGGER CONCEPTS

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

C-SPY and target systems

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

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



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 26*.

The ROM-monitor program

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

Third-party debuggers

You can use a third-party debugger together with the IAR toolchain as long as the third-party debugger can read ELF/DWARF, Intel-extended, or Motorola. For information about which format to use with a third-party debugger, see the user documentation supplied with that tool.

C-SPY plugin modules

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

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

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

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

EXECUTION MODES

IAR Embedded Workbench for Arm supports the 32-bit and 64-bit Arm architectures by means of execution modes.

In 32-bit mode refers to using IAR Embedded Workbench for Arm configured to generate and debug code for the T32/T and A32 instruction sets, either on an Armv4/5/6/7 core or in the AArch32 execution state on an Arm v8-A core. In 32-bit mode, you can use both the A32 and T32/T instruction sets and switch between them.

In 64-bit mode refers to using IAR Embedded Workbench for Arm configured to generate and debug code for the A64 instruction set in the AArch64 execution state on an Arm v8-A core. Code in 64-bit mode can call code in 32-bit mode, and that code can return back. However, the IAR translator tools do not support this switch being used in a single linked image. Switching between A32/T32/T code and A64 code must be performed by using several images. For example, an OS using 64-bit mode can start applications in either 64-bit or in 32-bit mode.

The AArch32 execution state is compatible with the Arm v7 architecture. The AArch32 execution state is emulated inside the AArch64 execution state.

C-SPY DRIVERS OVERVIEW

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

- Simulator
- CADI (Cycle Accurate Debug Interface)
- CMSIS-DAP probes

- The Renesas E2/E2Lite probes (with a Cortex-M device)
- GDB Server
- The GeneralPlus G+LINK probe
- I-jet and I-jet Trace debug probes
- J-Link/J-Trace probes
- Nuvoton Technology Corporation's Nu-Link debug adapter for Nuvoton's N76E Series MCUs. For information about the Nu-Link debug adapter, contact Nuvoton Technology Corporation.
- P&E Microcomputer Systems. For information about this driver, see the file `PEMicroSettings.pdf`, available in the `arm\doc` directory.
- ST-LINK JTAG/SWD probes (for ST Cortex-M devices only)
- TI MSP-FET probes
- TI Stellaris JTAG/SWD probes using FTDI or ICDI (for Stellaris Cortex devices only)
- TI XDS probes (XDS100v2, XDS100v3, XDS110, and XDS200)



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 507*.

Differences between the C-SPY drivers

This table summarizes the key differences between the Simulator, I-jet, J-Link/J-Trace, ST-LINK, TI XDS, CMSIS-DAP and TI MSP-FET drivers:

Feature	Simulator	I-jet	J-Link/ J-Trace	ST-LINK	TI XDS	CMSIS-DAP	TI MSP-FET
Code breakpoints	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Data breakpoints	Yes[3]	Yes	Yes	Yes[3]	Yes	Yes	Yes
Interrupt logging[1]	Yes[3]	Yes	Yes	Yes[3]	Yes	—	—
Data logging[1]	Yes[3]	Yes	Yes	Yes[3]	Yes	—	—
State logging[1]	—	—	—	—	Yes	—	Yes
Call stack trace[1]	Yes[3]	Yes	Yes	—	—	—	—
Event logging[1]	—	Yes	Yes	Yes[3]	Yes	—	—
Live watch[1]	Yes	Yes	Yes	Yes[3]	Yes	Yes	—
Cycle counter[1]	Yes	Yes	Yes	Yes[3]	Yes	Yes	—
Code coverage[1]	Yes[3]	Yes	Yes	Yes[3]	Yes	Yes	—
Data coverage	Yes[3]	—	—	—	—	—	—
Function /instruction profiler[1]	Yes[3]	Yes	Yes	Yes[3]	Yes	—	—
Trace[1]	Yes[3]	Yes	Yes	Yes[3]	Yes	—	—
Multicore debugging[1]	Yes[3]	Yes	Yes	Yes[4]	—	Yes[2]	—
Performance monitoring[1]	—	Yes	Yes	—	—	—	—
Power debugging[1]	—	Yes	Yes	Yes[3]	Yes	—	Yes

Table 3. Driver differences—Simulator, I-jet, J-Link/J-Trace, ST-LINK,TI XDS, etc

1 With specific requirements or restrictions, see the respective section in this documentation.

2 Limited support.

3 Not for all cores and devices.

4 See the manufacturer's documentation.

This table summarizes the key differences between the Simulator and other supported hardware debugger drives—GDB Server, TI Stellaris, CADI, Nu-Link, E2/E2Lite, and G+LINK:

Feature	Simulator	GDB Server	TI Stellaris	CADI	Nu-Link	E2/E2Lite	G+LINK
Code breakpoints	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Data breakpoints	Yes[3]	Yes	Yes	—	—	—	Yes
Interrupt logging[1]	Yes[3]	—	—	—	—	—	—
Data logging[1]	Yes[3]	—	—	—	—	—	—
State logging	—	—	—	—	—	—	—
Call stack trace[1]	Yes[3]	—	—	—	—	—	—
Event logging	—	—	—	—	—	—	—
Live watch	Yes	—	—	—	—	—	Yes
Cycle counter[1]	Yes	—	—	—	—	—	—
Code coverage[1]	Yes[3]	—	—	—	—	—	—
Data coverage	Yes[3]	—	—	—	—	—	—
Function/instruction profiler[1]	Yes[3]	—	—	—	—	—	—
Trace[1]	Yes[3]	—	—	—	—	—	—
Multicore debugging[1]	Yes[3]	—	—	—	—	—	—
Performance monitoring	—	—	—	—	—	—	—
Power debugging	—	—	—	—	—	—	—

Table 4. Driver differences—other drivers

1 With specific requirements or restrictions, see the respective section in this documentation.

2 Limited support.

3 Not for all cores and devices.

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.

Two simulator drivers

The C-SPY simulator uses one of two debugger drivers. Which driver depends on which core or device you have selected for your application project. For some cores and devices, the C-SPY simulator uses the IAR C-SPY simulator driver, a feature-rich debugger driver. For other cores and devices, the C-SPY Simulator uses the Imperas Instruction Set Simulator, a very fast debugger driver. Both simulators are fully integrated into the C-SPY framework.

You can see which simulator driver that C-SPY is using by inspecting the debug log. When the Imperas Instruction Set Simulator is used, a copyright statement from Imperas Software Ltd is printed to the log. If there is no such statement, C-SPY uses the IAR C-SPY simulator driver.

Supported features

The C-SPY simulator supports:

- Instruction-level simulation

- Memory configuration and validation
- Interrupt simulation (not supported by the Imperas driver)
- 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.

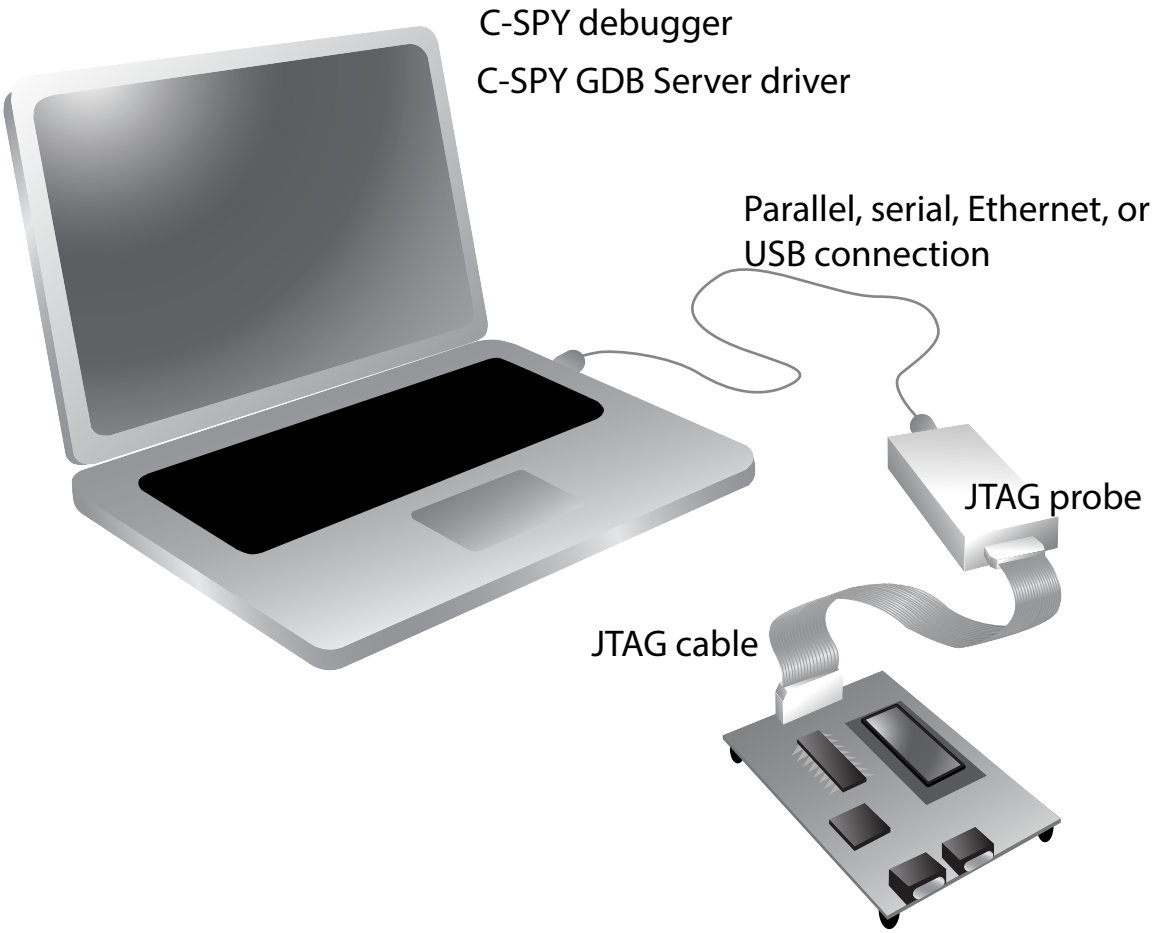
When a debug session is started, your application is automatically downloaded and programmed into target memory. You can disable this feature, if necessary.

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

Hardware installation

For best results, follow these steps.

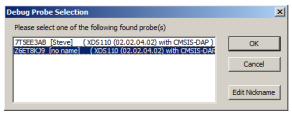
Recommended power-up sequence

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

1. Connect the probe to the target board.
2. Connect the USB cable to the debug probe.
3. Power up the debug probe, if it is not powered via USB.
4. Power up the target board, if it is not powered via the debug probe.
5. Start the C-SPY debug session.
6. If more than one debug probe is connected to your computer, the **Debug Probe Selection** dialog box is displayed. In the dialog box, select the probe to use and click **OK**. For more information, see *--drv_communication, page 423*.

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

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



USB driver installation

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

Installing the I-jet USB driver

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

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

Installing the I-jet Trace USB driver

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

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

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

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.

   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 version 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\Win_7-8` directory—`dpinst_amd64.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 using the `run_installer.bat` file which can be found in the `arm\drivers\ti-xds\win64` directory. It is recommended to choose the suggested installation directory. See also *TI XDS — Setup, page 503*.
3. Use the USB cable to connect the computer to the TI board.

Configuring the OpenOCD Server

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

Installing the TI MSP-FET USB driver

Before you can use the TI MSP-FET driver, the TI MSP-FET USB driver must be installed.

1. Install IAR Embedded Workbench for Arm.
2. The USB driver can be found in the product installation in the `arm\drivers\ti-mspfet\USB_CDC` directory—`DPInst64.exe`.
3. Use the USB cable to connect to the TI MSP-FET probe.

Installing the PE Micro USB driver

To install the USB driver for the PE Micro driver manually, navigate to the `arm\drivers\pemicro` directory and start the `PEDrivers_install.exe` application. This will install the USB driver. For information about this driver, see the file `PEMicroSettings.pdf`, available in the `arm\doc` directory.

Installing the Nu-Link USB driver

To install the USB driver for the Nu-Link driver manually, navigate to the `arm\drivers\Nu-Link` directory and start the `Nu-Link_USB_Driver.exe` application. This will install the USB driver. For information about this driver, see the file `NuMicro Cortex-M IAR ICE driver user manual.pdf` available in the `arm\bin\Nu-Link` directory.

Installing the E2/E2Lite USB driver

To install the USB driver for the Renesas E2 C-SPY driver manually, navigate to the `arm\drivers\RenesasE2` directory and start the `USB_Driver_x64_for_Renesas_MCU_Tools.exe` application. This will install the USB driver. For information about this probe, see the documentation from Renesas supplied with the probe.

Getting started using C-SPY

Contents

Setting up C-SPY 33
Setting up for debugging 33
Executing from reset 34
Using a setup macro file 34
Selecting a device description file 35
Loading plugin modules 35
Starting C-SPY 35
Starting a debug session 35
Loading executable files built outside of the IDE 35
Starting a debug session with source files missing 36
Loading multiple debug images 36
Editing in C-SPY windows 37
Adapting for target hardware 38
Modifying a device description file 38
Initializing target hardware before C-SPY starts 38
Remapping memory 38
Using predefined C-SPY macros for device support 39
Debug authentication 39
An overview of the debugger startup 40
Debugging code in flash 40
Debugging code in RAM 41
Reference information on starting C-SPY 41
C-SPY Debugger main window 41
Images window 46
Get Alternative File dialog box 47

SETTING UP C-SPY

Setting up for debugging

1. Install a USB driver or some other communication driver, if your C-SPY driver requires it.
   For more information, see:

   - *Installing the I-jet USB driver, page 31*
   - *Installing the I-jet Trace USB driver, page 31*
   - *Installing the J-Link USB driver, page 31*
   - *Installing the ST-LINK USB driver for ST-LINK version 2, page 31*
   - *Installing the TI Stellaris USB driver, page 32*
   - *Installing the TI XDS USB driver, page 32*
   - *Configuring the OpenOCD Server, page 32*
   - *Installing the TI MSP-FET USB driver, page 32*
   - *Installing the PE Micro USB driver, page 32*
   - *Installing the Nu-Link USB driver, page 32*
   - *Installing the E2/E2Lite USB driver, page 32*

2. Choose **Project>Options>General>Target** and make sure that the device you are creating the application for is selected.
3. In the **Category** list, go to the **Debugger>Setup** page and select the C-SPY driver that matches your debugger system—simulator or hardware debugger system.
4. In the **Category** list, click the appropriate C-SPY driver category and make your settings. For information about these options, see *Debugger options, page 465*.
5. Click **OK**.
6. 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 38*.

Now you are ready to start C-SPY.

Executing from reset

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

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

If you leave the check box empty, the program counter will contain the regular hardware reset address at each reset. The reset address is set by C-SPY.

If no breakpoints are available when C-SPY starts, a warning message notifies you that single stepping will be required and that this is time consuming. You can then continue execution in single-step mode or stop at the first instruction. If you choose to stop at the first instruction, the debugger starts executing with the `PC` (program counter) at the default reset location instead of the location you typed in the **Run to** box.



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 330*.

For an example of how to use a setup macro file, see *Initializing target hardware before C-SPY starts, page 38*.

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 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 38*.

To override the default device description file:

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



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

Loading plugin modules

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

For more information, see *Plugins, page 474*.

STARTING C-SPY

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

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 select the template **Externally built executable**. Click **OK** and specify a project name in the dialog box.
2. To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the file type drop-down list. Locate the executable file.
3.  To start the executable file, click the **Download and Debug** button. The project can be reused whenever you rebuild your executable file.

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

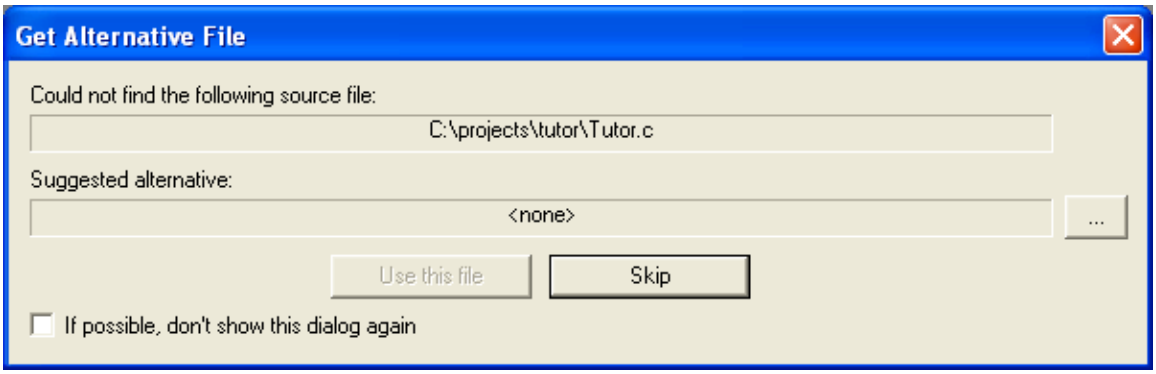


To make it possible for the IDE to find and display all source files referred to from the externally built binary file, a source file resolution system exists, based on aliases. For more information, see the IDE Project Management and Building Guide for Arm.

Starting a debug session with source files missing

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

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



Typically, you can use the dialog box like this:

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

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

For more information, see *Get Alternative File dialog box, page 47*.

Loading multiple debug images

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

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

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

To load additional debug images at C-SPY startup:

1. Choose **Project>Options>Debugger>Images** and specify up to three additional debug images to be loaded. For more information, see *Images, page 469*.
2. Start the debug session.

   To load additional debug images at a specific moment:

   Use the `__loadImage` system macro and execute it using either one of the methods described in *Using C-SPY macros, page 332*.

   To display a list of loaded debug images:

   Choose **Images** from the **View** menu. The **Images** window is displayed, see *Images window, page 46*.

Editing in C-SPY windows

You can edit the contents of these windows:

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

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

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

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

```
myArray;3
```

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

```
myPtr;3
```

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

```
myArray;5,10
```

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

```
myPtr;5,10
```



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

ADAPTING FOR TARGET HARDWARE

See also *Memory configuration for C-SPY hardware debugger drivers, page 122*.

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 35*. Device description files 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 306*.

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 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 35*.

Initializing target hardware before C-SPY starts

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

1. Create a new text file and define your macro function.
   By using the built-in `execUserPreload` setup macro function, your macro function will be executed directly after the communication with the target system is established but before C-SPY downloads your application.
   For example, a macro that enables external SDRAM could look like this:

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

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

2. Save the file with the filename extension `mac`.
3. Before you start C-SPY, choose **Project>Options>Debugger** and click the **Setup** tab.
4. Select the **Use macro file** option, and choose the macro file you just created.
   Your setup macro will now be loaded during the C-SPY startup sequence.

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 Microchip 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, 0xFFFFFF00, "Memory");
}
```



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 332*. For information about the macro functions used, see *Reference information on C-SPY system macros, page 343*.

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.

Debug authentication

The purpose of debug authentication is to reduce the risk that an attacker uses debug capabilities to compromise the target system. IAR Embedded Workbench supports debug authentication using the open-source Secure Debug Manager (SDM) mechanism. If the target device has been locked reversibly, the debug authentication mechanism unlocks the device for debugging, if you provide the correct credentials.

If authentication is enabled, it is performed when the debug session starts. The authentication is revoked by a target reset. The type of reset that revokes the authentication is hardware-specific.

When authentication is configured, IAR Embedded Workbench tries to determine whether it is actually needed so that the authentication procedure is not executed if the device is not locked. For example, if a power-on-reset is what revokes the authentication, restarting a debug session does not require a new authentication if no power cycle has taken place. If the device has never been locked, or has been unlocked in such a way that it does not go back to a locked state, no authentication is required either.

Any input dialog boxes displayed during the authentication process come from the SDM library used to drive the authentication process, not from IAR Embedded Workbench.

The SDM mechanism requires a library that implements the authentication protocol in terms of the SDM interface.



Authentication is only available if the hardware supports it, if the correct library, corresponding to the hardware, is used, and if the hardware debugger driver supports it.

AN OVERVIEW OF THE DEBUGGER STARTUP

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

For more information about C-SPY system macros, see *C-SPY macros, page 328*.

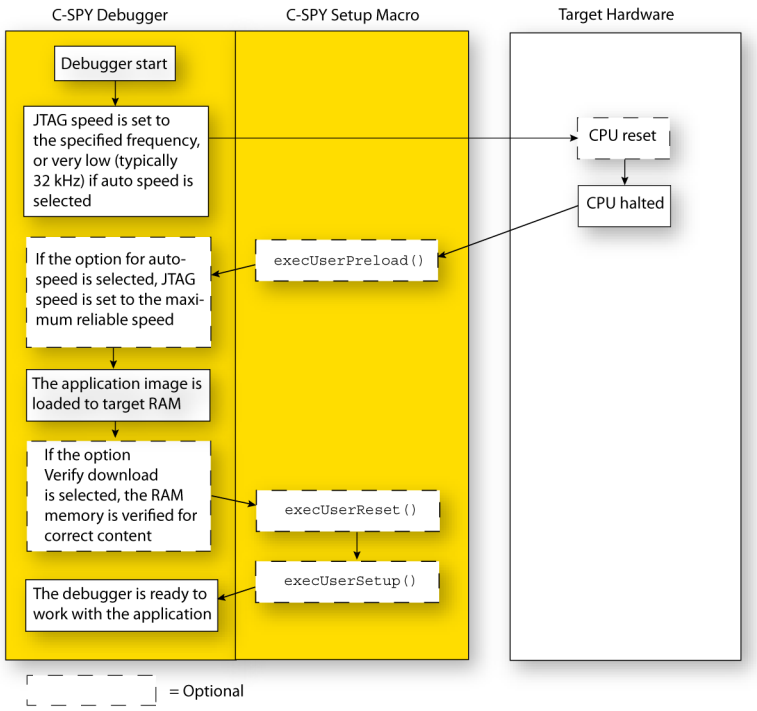
Debugging code in flash

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:



REFERENCE INFORMATION ON STARTING C-SPY

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
- A special multicore debugging toolbar
- Several windows and dialog boxes specific to C-SPY

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

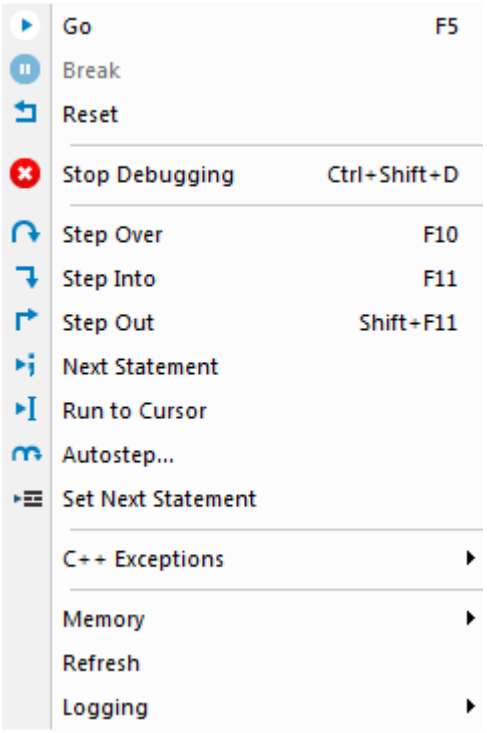
Menu bar

These menus are available during a debug session:

Debug	Provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.
C-SPY driver menu	Provides commands specific to a C-SPY driver. The driver-specific menu is only available when the driver is used. For information about the driver-specific menu commands, see *Reference information on C-SPY driver menus, page 509*.
Disassembly	Provides commands for controlling how the disassembler operates.

Debug menu

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



These commands are available:

Go (F5)

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

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

Break

Stops the application execution.

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

Reset

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

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

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

Stop Debugging (Ctrl+Shift+D)



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

Step Over (F10)



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

Step Into (F11)



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

Step Out (Shift+F11)



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

Next Statement



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

Run to Cursor



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

Autostep



Displays a dialog box where you can customize and perform autostepping, see *Autostep settings dialog box, page 66*.

Set Next Statement



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

C++ Exceptions>Break on Throw

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

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

C++ Exceptions>Break on Uncaught Exception

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

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

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 129*.

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 130*.

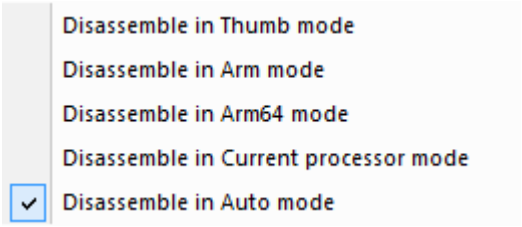
Refresh

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

Logging>Set Terminal I/O Log file	Displays a dialog box where you can choose to log simulated target access communication to a file. You can select the destination of the log file. See *Terminal I/O Log File dialog box, page 62*.

Disassembly menu

The **Disassembly** menu is available when C-SPY is running. This menu provides commands for controlling how the disassembler operates.



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

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 Arm64 mode	Disassembles your application in 64-bit 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 55*.

C-SPY windows

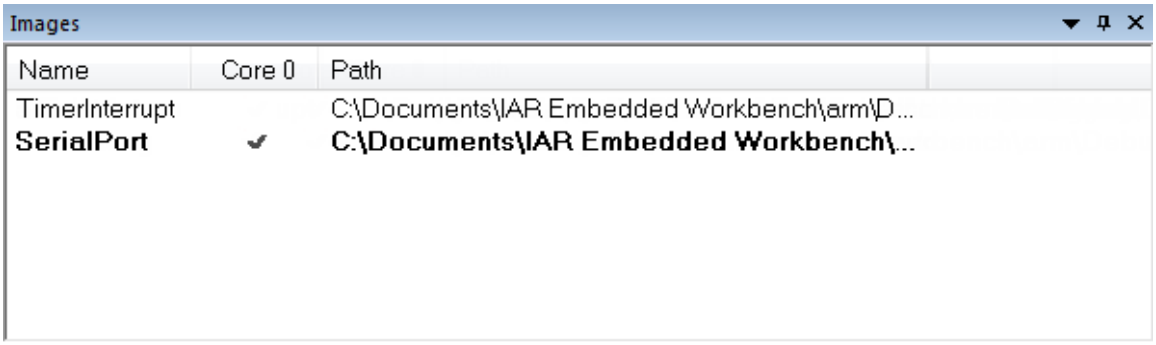
Depending on the C-SPY driver you are using, these windows specific to C-SPY are available during a debug session:

- *C-SPY Debugger main window, page 41*
- *Disassembly window, page 55*
- *Memory window, page 126*
- *Symbolic Memory window, page 132*
- *Registers window, page 138*
- *Watch window, page 76*
- *Locals window, page 74*
- *Auto window, page 71*
- *Live Watch window, page 78*
- *Quick Watch window, page 84*
- *Statics window, page 81*
- *Call Stack window, page 59*
- *Trace window, page 170*
- *Function Trace window, page 178*
- Timeline window, see *Reference information on application timeline, page 200*
- *Terminal I/O window, page 61*
- *Code Coverage window, page 234*
- *Function Profiler window, page 229*
- *Performance Monitoring window, page 240*
- *Images window, page 46*
- *Stack window, page 135*
- *Symbols window, page 87*

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

Images window

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



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

Normally, a source application consists of a single debug image that you debug. However, you can also load additional images. This means that the complete debuggable unit consists of several debug images. See also *Loading multiple debug images, page 36*.

Requirements

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

Display area

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

This area lists the loaded debug images in these columns:

Name	The name of the loaded debug image.
Core *N*	Double-click in this column to toggle using debug information from the debug image when that core is in focus.
Path	The path to the loaded debug image.

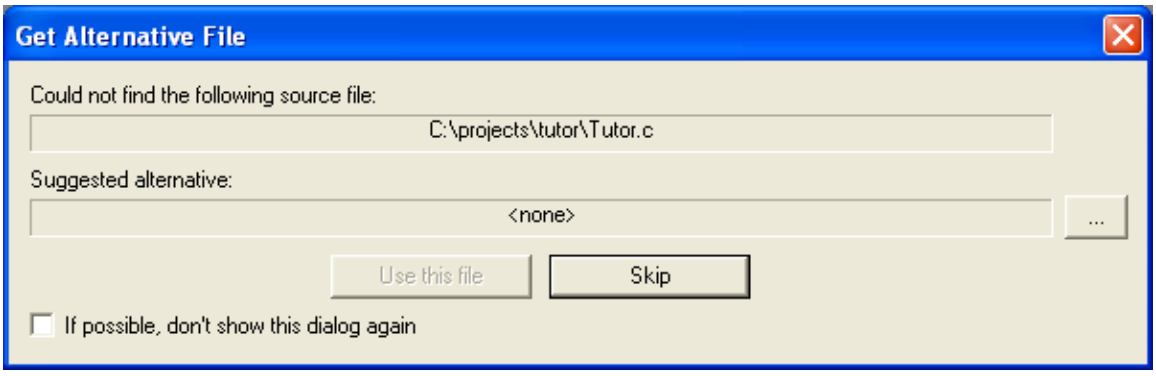
Related information

For related information, see:

- *Loading multiple debug images, page 36*
- *Images, page 469*
- *__loadImage, page 368*

Get Alternative File dialog box

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



See also *Starting a debug session with source files missing, page 36*.

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 36*.

Executing your application

Contents

Introduction to application execution 49
Briefly about application execution 49
Source and disassembly mode debugging 49
Single stepping 50
Troubleshooting slow stepping speed 51
Running the application 52
Highlighting 52
Viewing the call stack 53
Terminal input and output 54
Debug logging 54
Reference information on application execution 54
Disassembly window 55
Call Stack window 59
Terminal I/O window 61
Terminal I/O Log File dialog box 62
Debug Log window 63
Fault exception viewer window 64
Report Assert dialog box 65
Autostep settings dialog box 66

INTRODUCTION TO APPLICATION EXECUTION

Briefly about application execution

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

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

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

Source and disassembly mode debugging

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

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

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

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

Single stepping

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

There are several factors that can slow down the stepping speed. If you find it too slow, see *Troubleshooting slow stepping speed, page 51* 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 66*.

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

Consider this example and assume that the previous step has taken you to the `f(i)` function call (highlighted):

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) + g(n-3);
 return value;
}
int main()
{
  ...
  f(i);
  value ++;
}
```

Step Into

While stepping, you typically consider whether to step into a function and continue stepping inside the function or subroutine. The **Step Into** command takes you to the first step point within the subroutine `g(n-1)`:

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) + g(n-3);
 return value;
}
```

The **Step Into** command executes to the next step point in the normal flow of control, regardless of whether it is in the same or another function.

Step Over

The **Step Over** command executes to the next step point in the same function, without stopping inside called functions. The command would take you to the `g(n-2)` function call, which is not a statement on

its own but part of the same statement as `g(n-1)`. Thus, you can skip uninteresting calls which are parts of statements and instead focus on critical parts:

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) + g(n-3);
 return value;
}
```

Next Statement

The **Next Statement** command executes directly to the next statement, in this case `return value`, allowing faster stepping:

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) + g(n-3);
 return value;
}
```

Step Out

When inside the function, you can—if you wish—use the **Step Out** command to step out of it before it reaches the exit. This will take you directly to the statement immediately after the function call:

```
extern int g(int);
int f(int n)
{
 value = g(n-1) + g(n-2) g(n-3);
 return value;
}
int main()
{
  ...
  f(i);
  value ++;
}
```

The possibility of stepping into an individual function that is part of a more complex statement is particularly useful when you use C code containing many nested function calls. It is also very useful for C++, which tends to have many implicit function calls, such as constructors, destructors, assignment operators, and other user-defined operators.

This detailed stepping can in some circumstances be either invaluable or unnecessarily slow. For this reason, you can also step only on statements, which means faster stepping.

Troubleshooting slow stepping speed

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

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

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

  For more information, see *Breakpoints in the C-SPY hardware debugger drivers, page 93* and *Breakpoint consumers, page 93*.

- Disable trace data collection, using the **Enable/Disable** button in both the **Trace** and the **Function Profiling** windows. Trace data collection might slow down stepping because the collected trace data is processed after each step. Note that it is not sufficient to just close the corresponding windows to disable trace data collection.
- Choose to view only a limited selection of SFR registers. You can choose between two alternatives. Either type `#SFR_name` (where `SFR_name` reflects the name of the SFR you want to monitor) in the **Watch** window, or create your own filter for displaying a limited group of SFRs in the **Registers** window. Displaying many SFR registers might slow down stepping because all registers must be read from the hardware after each step. See *Defining application-specific register groups, page 123*.
- Close the **Memory** and **Symbolic Memory** windows if they are open, because the visible memory must be read after each step and that might slow down stepping.
- Close any window that displays expressions such as **Watch**, **Live Watch**, **Locals**, **Statics** if it is open, because all these windows read memory after each step and that might slow down stepping.
- Close the **Stack** window if it is open. Choose **Tools>Options>Stack** and disable the **Enable graphical stack display and stack usage tracking** option if it is enabled.
- If possible, increase the communication speed between C-SPY and the target board/emulator.

Running the application

Go

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

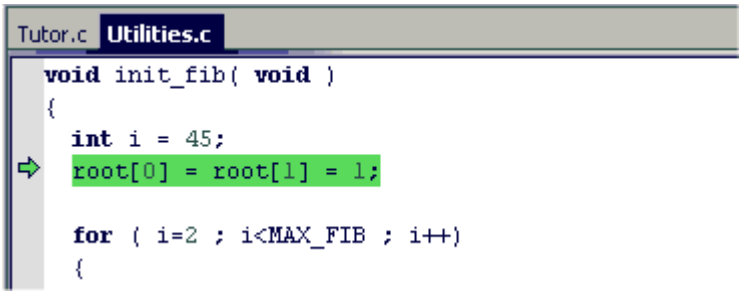
If you are using symmetric multicore debugging, the **Go** command starts only the core in focus.

Run to Cursor

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

Highlighting

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



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

Occasionally, you will notice that a statement in the source window is highlighted using a pale variant of the normal highlight color. This happens when the program counter is at an assembler instruction which

is part of a source statement but not exactly at a step point. This is often the case when stepping in the **Disassembly** window. Only when the program counter is at the first instruction of the source statement, the ordinary highlight color is used.

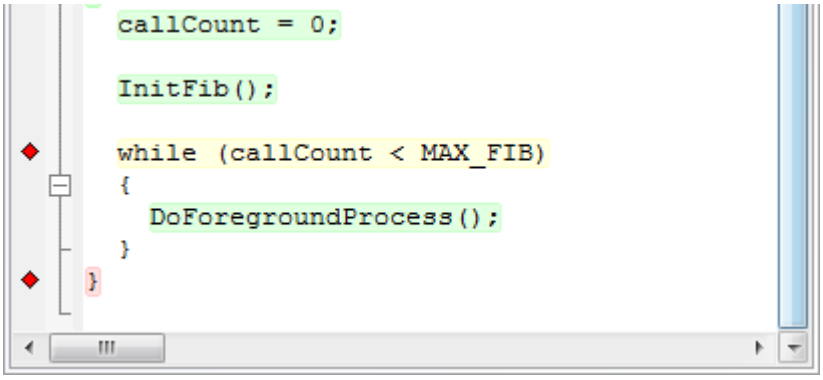
Code coverage

From the context menu in the **Code Coverage** window, you can toggle highlight colors and icons in the editor window that show code coverage analysis for the source code, see *Code Coverage window, page 234*.

These are the colors and icons that are used:

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

This figure illustrates all three code coverage highlight colors:



Viewing the call stack

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



Typically, this is useful for two purposes:

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

The **Call Stack** window shows a list of function calls, with the current function at the top. When you inspect a function in the call chain, the contents of all affected windows are updated to display the state of that particular call frame. This includes the editor, **Locals**, **Register**, **Watch**, and **Disassembly** windows. A function would normally not make use of all registers, so these registers might have undefined states and be displayed as dashes (---).

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

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

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



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

Terminal input and output

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



This facility is useful in two different contexts:

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

For more information, see *Terminal I/O window, page 61* and *Terminal I/O Log File dialog box, page 62*.

Debug logging

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



It can sometimes be convenient to log the information to a file where you can easily inspect it, see *Debug Log window, page 63*. The two main advantages are:

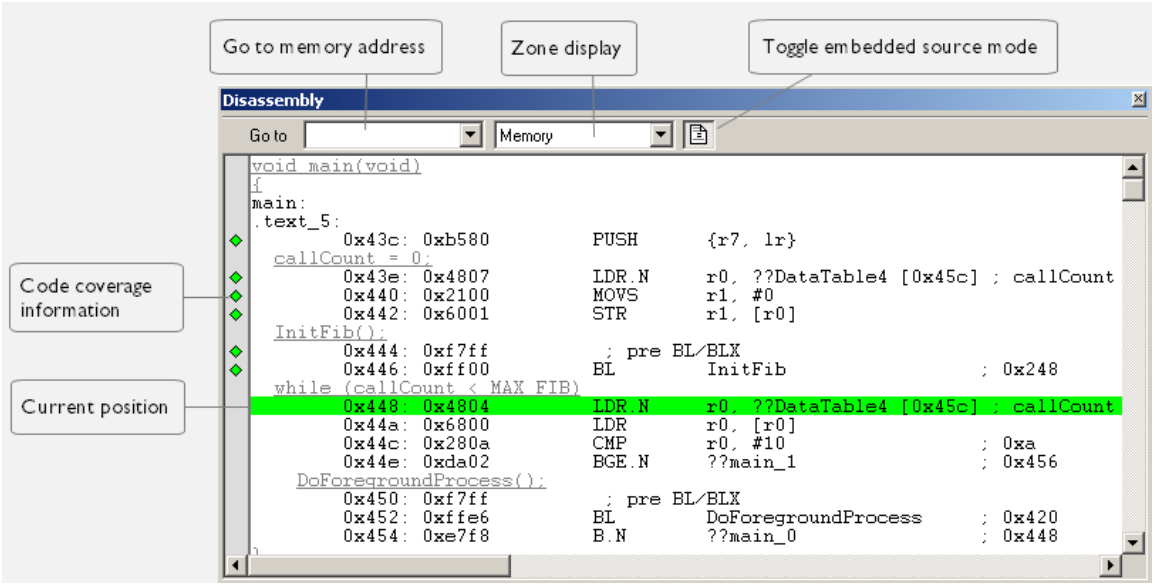
- The file can be opened in another tool, for instance an editor, so you can navigate and search within the file for particularly interesting parts.
- The file provides history about how you have controlled the execution, for instance, which breakpoints that have been triggered etc.

REFERENCE INFORMATION ON APPLICATION EXECUTION

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 color in disassembly window** option.



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

See also *Source and disassembly mode debugging, page 49*.

Requirements

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

Toolbar

The toolbar contains:

Go to	The memory location or symbol you want to view.
Zone	Selects a memory zone, see *C-SPY memory zones, page 121*.
Toggle Mixed-Mode	Toggles between displaying only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Display area

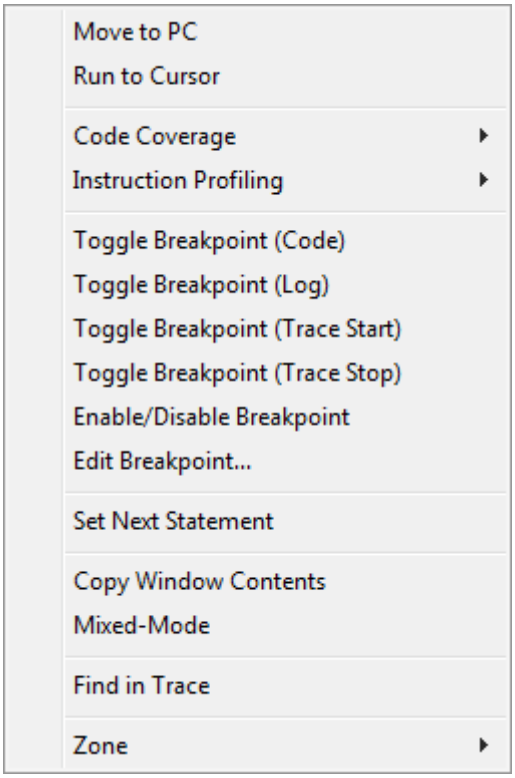
The display area shows the disassembled application code. This area contains these graphic elements:

Green highlight color	Indicates the current position, that is the next assembler instruction to be executed. To move the cursor to any line in the **Disassembly** window, click the line. Alternatively, move the cursor using the navigation keys.
Yellow highlight color	Indicates a position other than the current position, such as when navigating between frames in the **Call Stack** window or between items in the **Trace** window.
Red dot	Indicates a breakpoint. Double-click in the gray left-side margin of the window to set a breakpoint. For more information, see *Breakpoints, page 90*.
Green diamond	Code coverage icon—indicates code that has been executed.
Red diamond	Code coverage icon—indicates code that has *not* been executed.
Red/yellow diamond (red top/yellow bottom)	Code coverage icon—indicates a branch that is *never* taken.
Red/yellow diamond (red left side/yellow right side)	Code coverage icon—indicates a branch that is *always* taken.

If instruction profiling has been enabled from the context menu, an extra column in the left-side margin appears with information about how many times each instruction has been executed. For debug probes that support it, C-SPY can capture full instruction trace in real time.

Context menu

This context menu is available:



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

These commands are available:

Move to PC Displays code at the current program counter location.

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

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

Enable	Toggles code coverage on or off.
Show	Toggles the display of code coverage on or off. Code coverage is indicated by a red, green, and red/yellow diamonds in the left margin.
Clear	Clears all code coverage information.
Next Different Coverage >	Moves the insertion point to the next line in the window with a different code coverage status than the selected line.
Previous Different Coverage <	Moves the insertion point to the closest preceding line in the window with a different code coverage status than the selected line.

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

Enable	Toggles instruction profiling on or off.

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

Toggle Breakpoint (Code) Toggles a code breakpoint. Assembler instructions and any corresponding label at which code breakpoints have been set are highlighted in red. For more information, see *Code breakpoints dialog box, page 102*.

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 106*.

Toggle Breakpoint (Trace Start) Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Start Trigger breakpoint dialog box, page 179*.

Toggle Breakpoint (Trace Stop) Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Stop Trigger breakpoint dialog box, page 180*.

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

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

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

Copy Window Contents Copies the selected contents of the **Disassembly** window to the clipboard.

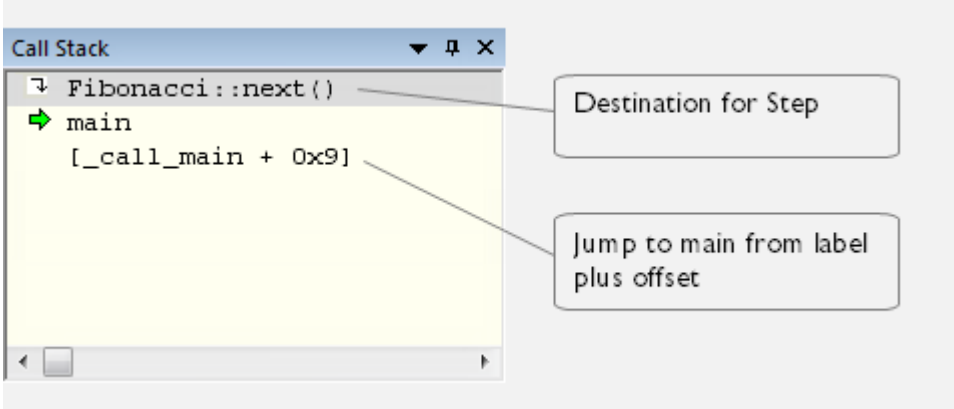
Mixed-Mode Toggles between showing only disassembled code or disassembled code together with the corresponding source code. Source code requires that the corresponding source file has been compiled with debug information.

Find in Trace Searches the contents of the **Trace** window for occurrences of the given location—the position of the insertion point in the source code—and reports the result in the **Find in Trace** window. This menu command requires support for Trace in the C-SPY driver you are using, see *Differences between the C-SPY drivers, page 27*.

Zone Selects a memory zone, see *C-SPY memory zones, page 121*.

Call Stack window

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



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

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

See also *Viewing the call stack, page 53*.

Requirements

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

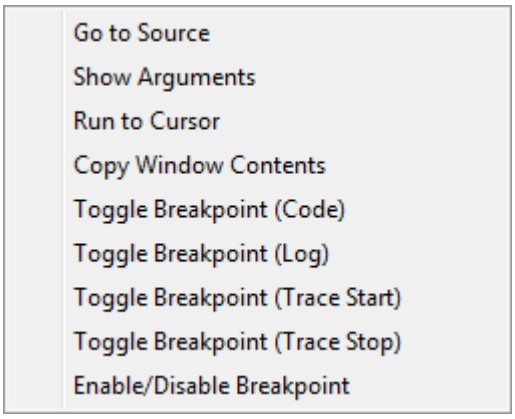
Display area

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

function(*values*)***	A C/C++ function with debug information. Provided that **Show Arguments** is enabled, *values* is a list of the current values of the parameters, or empty if the function does not take any parameters. ***, if present, indicates that the function has been inlined by the compiler. For information about function inlining, see the *IAR C/C++ Development Guide for Arm*.
[*label + offset*]	An assembler function, or a C/C++ function without debug information.
<*exception_frame*>	An interrupt.

Context menu

This context menu is available:

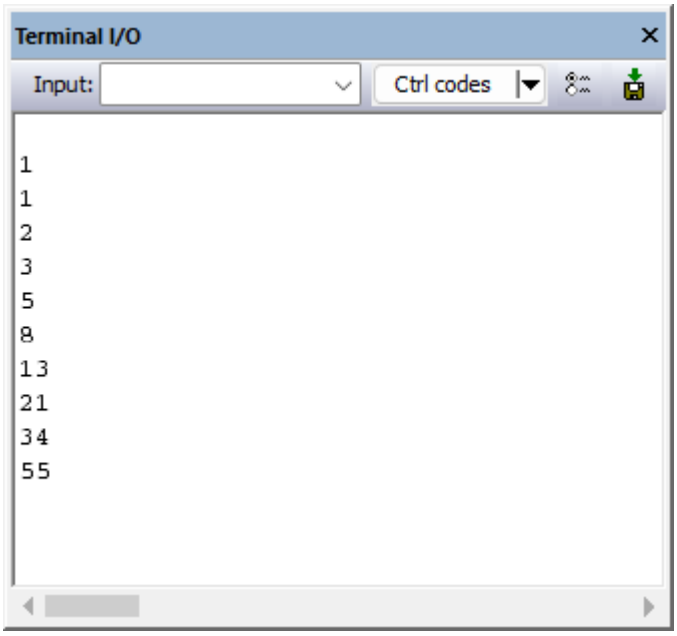


These commands are available:

Go to Source	Displays the selected function in the **Disassembly** or editor windows.
Show Arguments	Shows function arguments.
Run to Cursor	Executes until return to the function selected in the call stack.
Copy Window Contents	Copies the contents of the **Call Stack** window and stores them on the clipboard.
Toggle Breakpoint (Code)	Toggles a code breakpoint.
Toggle Breakpoint (Log)	Toggles a log breakpoint.
Toggle Breakpoint (Trace Start)	Toggles a Trace Start breakpoint. When the breakpoint is triggered, trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports it.
Toggle Breakpoint (Trace Stop)	Toggles a Trace Stop breakpoint. When the breakpoint is triggered, trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports it.
Enable/Disable Breakpoint	Enables or disables the selected breakpoint.

Terminal I/O window

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



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

To use this window, you must:

- 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 `stdout` of your application is routed via SWO. See *SWO Configuration dialog box, page 167*, 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 `arm\src\debugger\dcc\DCC_Write.c` to your project. `DCC_write.c` overrides the library function `write`. Functions such as `printf` can then be used to output text 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

See also *Terminal input and output, page 54*.

Requirements

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

Toolbar

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`.



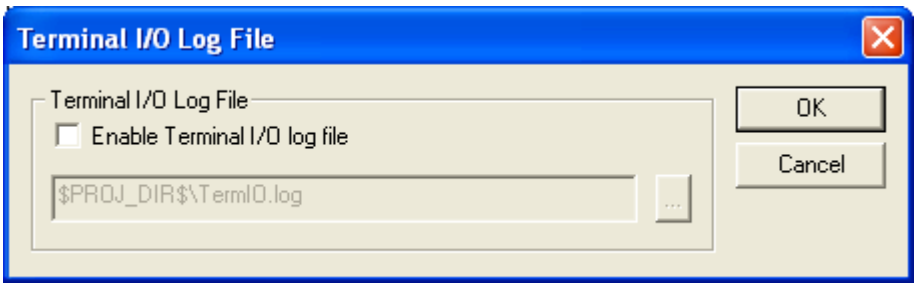
Opens the **IDE Options** dialog box where you can set options for terminal I/O. For information about the options available in this dialog box, see *Terminal I/O options* in *IDE Project Management and Building Guide for Arm*.



Opens the **Terminal I/O Log File** dialog box, where you can configure and toggle logging on/off, see *Terminal I/O Log File dialog box, page 62*.

Terminal I/O Log File dialog box

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



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

See also *Terminal input and output, page 54*.

Requirements

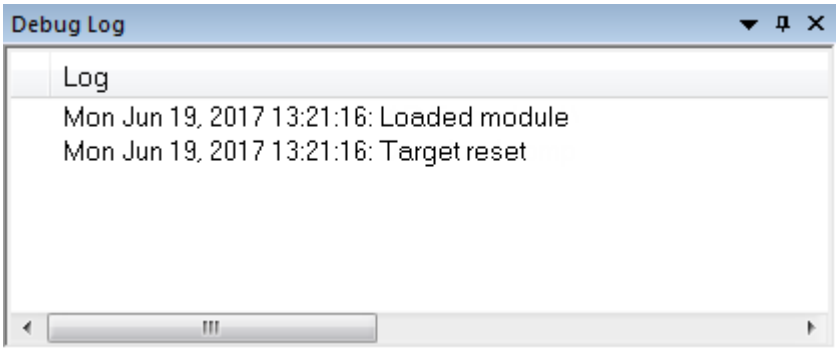
Can be used with all C-SPY debugger drivers and debug probes.

Terminal I/O Log File

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

Debug Log window

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



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

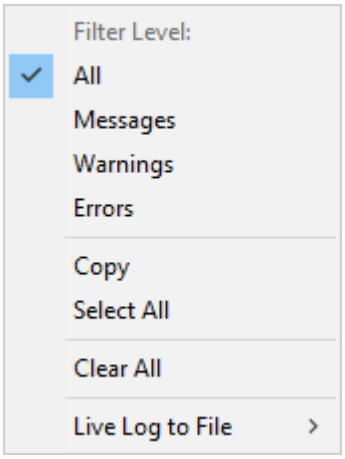
See also *Debug logging, page 54*.

Requirements

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

Context menu

This context menu is available:

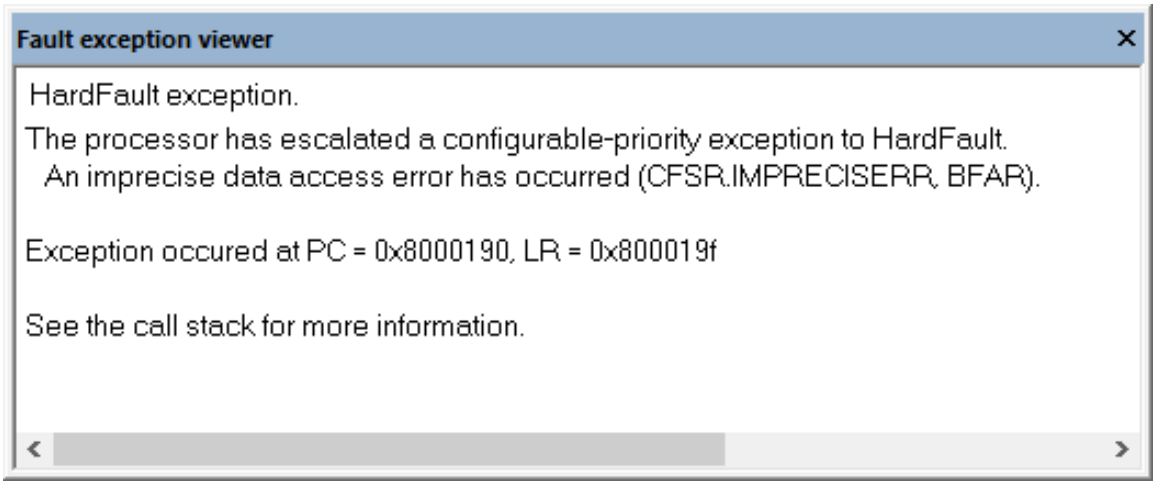


These commands are available:

All	Shows all messages sent by the debugging tools and drivers.
Messages	Shows all C-SPY messages.
Warnings	Shows warnings and errors.
Errors	Shows errors only.
Copy	Copies the contents of the window.
Select All	Selects the contents of the window.
Clear All	Clears the contents of the window.
Live Log to File	Displays a submenu with commands for writing the debug messages to a log file and setting filter levels for the log.

Fault exception viewer window

The **Fault exception viewer** window is available by choosing **View>Fault exception viewer**. The window is automatically opened when the execution stops and a fault exception was encountered.



Requirements

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

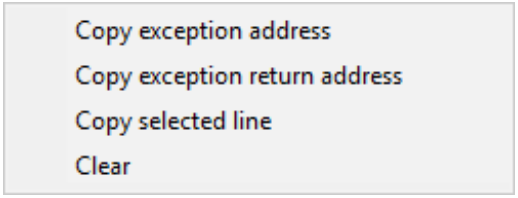
To display exception messages in this window, the selected device must be a Cortex-M device.

Display area

This area contains a description of the most recent fault exception that was encountered during application execution.

Context menu

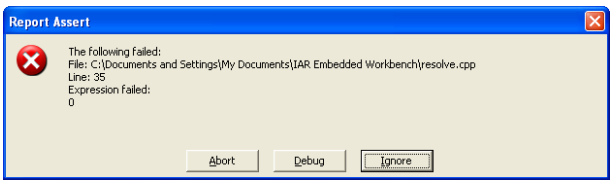
This context menu is available:



Copy exception address	Copies the address that the program counter, `PC`, had at the time of the exception. The address is copied to the clipboard.
Copy exception return address	Copies the address that the link register, `LR`, had at the time of the exception. The address is copied to the clipboard.
Copy selected line	Copies the selected line to the clipboard.
Clear	Clears the messages displayed in the **Fault exception viewer** window.

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

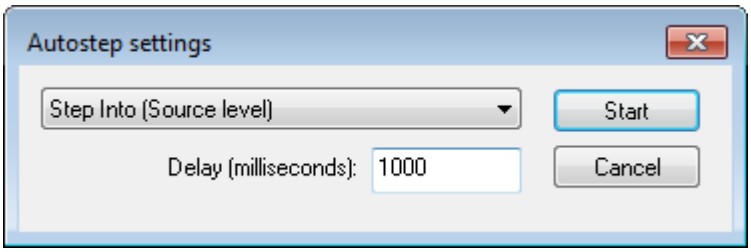
C-SPY stops the execution of the application and returns control to you.

Ignore

The assertion is ignored and the application continues to execute.

Autostep settings dialog box

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



Use this dialog box to configure autostepping.

Select the step command you want to automate from the drop-down menu. The step will be performed with the specified interval. For a description of the available step commands, see *Single stepping, page 50*. You can stop the autostepping by clicking the Break button on the debug toolbar.

Requirements

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

Delay (milliseconds)

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



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

Variables and expressions

Contents

Introduction to working with variables and expressions 67
Briefly about working with variables and expressions 67
C-SPY expressions 68
Limitations on variable information 70
Working with variables and expressions 70
Using the windows related to variables and expressions 70
Viewing assembler variables 71
Reference information on working with variables and expressions 71
Auto window 71
Locals window 74
Watch window 76
Live Watch window 78
Statics window 81
Quick Watch window 84
Symbols window 87
Resolve Symbol Ambiguity dialog box 89

INTRODUCTION TO WORKING WITH VARIABLES AND EXPRESSIONS

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

Briefly about working with variables and expressions

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

- Tooltip watch—in the editor window—provides the simplest way of viewing the value of a variable or more complex expressions. Just point at the variable with the mouse pointer. The value is displayed next to the variable.
- The **Auto** window displays a useful selection of variables and expressions in, or near, the current statement. The window is automatically updated when execution stops.
- The **Locals** window displays the local variables, that is, auto variables and function parameters for the active function. The window is automatically updated when execution stops.
- The **Watch** window allows you to monitor the values of C-SPY expressions and variables. The window is automatically updated when execution stops.
- The **Live Watch** window repeatedly samples and displays the values of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.
- The **Statics** window displays the values of variables with static storage duration. The window is automatically updated when execution stops.
- The **Macro Quicklaunch** window and the **Quick Watch** window give you precise control over when to evaluate an expression.
- The **Symbols** window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.

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

- The **Data Log** window and the **Data Log Summary** window display logs of accesses to up to four different memory locations you choose by setting data log breakpoints. Data logging can help you locate frequently accessed data. You can then consider whether you should place that data in more efficient memory.
- The **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.

For more information about these windows, see *The application timeline, page 195*.

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, page 68*
- *Assembler symbols, page 69* (register names and assembler labels)
- *C-SPY macro functions, page 69*
- *C-SPY macro variables, page 69*

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
#PC32
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. For global variables (variables defined in file scope), you can use the notation *module*`\`*variable*—the same syntax that is used to denote variables in the **Statics** window. For example, if there are two source files `source_a.c` and `source_b.c`, each with an identical `static int var = 0;` definition, you can write `source_a\var` or `source_b\var` to specify a specific variable.



On 64-bit architectures, the expression `#PC` becomes ambiguous in C-SPY macros. Therefore, when you debug a 64-bit MCU, you must use `#PC32` for AArch32 and `#PC64` for AArch64 in C-SPY macros. When debugging a 32-bit MCU, use `#PC`.

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.



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 __mattr *)
```

However, this line will be accepted:

```
sizeof(unsigned char __mattr *)
```

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 38*.

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

Example	What it does
`#PC32++`	Increments the value of the program counter.
`myVar = #SP`	Assigns the current value of the stack pointer register to your C-SPY variable.
`myVar = #label`	Sets `myVar` to the value of an integer at the address of `label`.
`myptr = &#label7`	Sets `myptr` to an `int *` pointer pointing at `label7`.

Table 6. C-SPY assembler symbols expressions

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

Example	What it does
`#PC32`	Refers to the program counter.
`#´PC32´`	Refers to the assembler label `PC32`.

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

Which processor-specific symbols are available by default can be seen in the **Registers** window, using the `CPU Registers` register group. See *Registers window, page 138*.

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 331*.

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 335*.

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.



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

Limitations on variable information

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

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

Effects of optimizations

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

Consider this example:

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

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

```
Unavailable
```

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

WORKING WITH VARIABLES AND EXPRESSIONS

See also *Analyzing your application's timeline, page 197*.

Using the windows related to variables and expressions

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

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

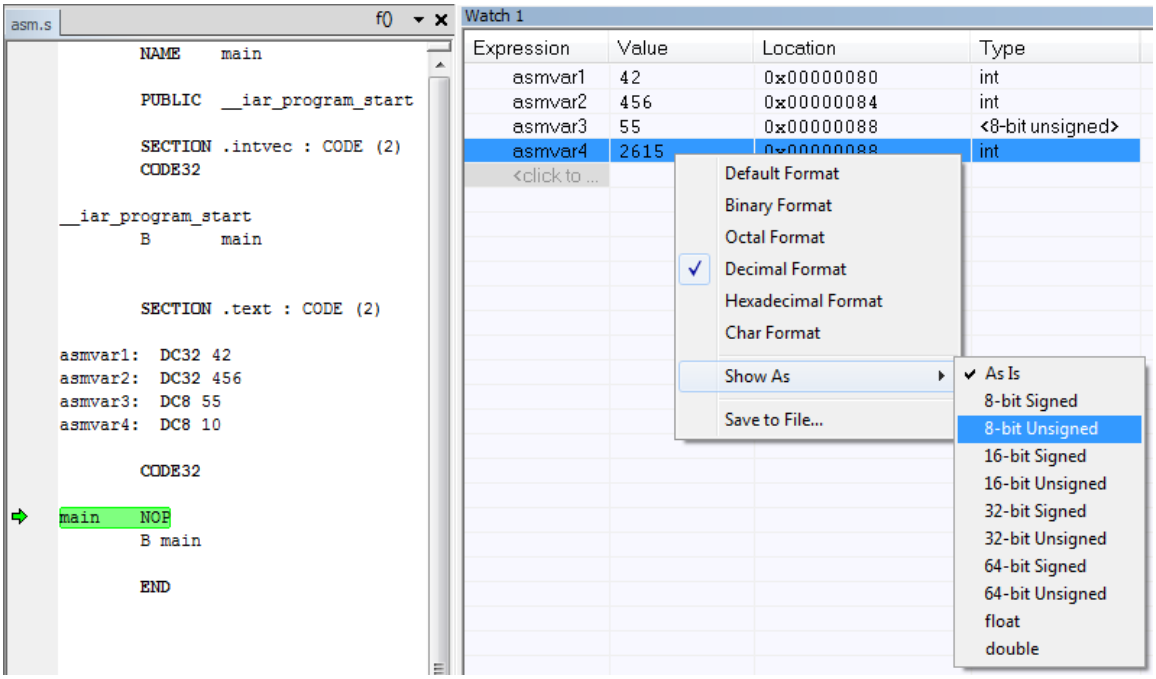
For text that is too wide to fit in a column—in any of these windows, except the **Trace** window—and thus is truncated, just point at the text with the mouse pointer and tooltip information is displayed.

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

Viewing assembler variables

An assembler label does not convey any type information at all, which means C-SPY cannot easily display data located at that label without getting extra information. To view data conveniently, C-SPY by default treats all data located at assembler labels as variables of type `int`. However, in the **Watch**, **Live Watch**, and **Quick Watch** windows, you can select a different interpretation to better suit the declaration of the variables.

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



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

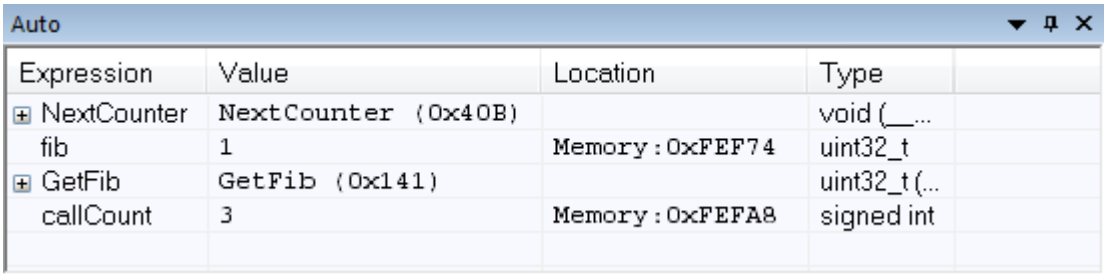
REFERENCE INFORMATION ON WORKING WITH VARIABLES AND EXPRESSIONS

See also:

- *Reference information on trace, page 161* for trace-related reference information
- *Macro Quicklaunch window, page 404*

Auto window

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



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

See also *Editing in C-SPY windows, page 37*.

Requirements

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

Display area

This area contains these columns:

Expression The name of the variable. This column is not editable.

A name in square brackets, for example `[fieldname]`, is either the name of a base class for a struct/class object or—if shown in gray and *italic*)—a pointer to the full object when of a class type derived from the original class.

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.

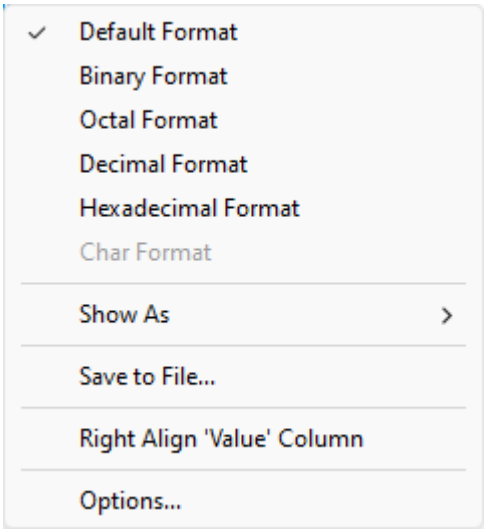
This column is editable.

Location The location in memory where this variable is stored.

Type The data type of the variable.

Context menu

This context menu is available:



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

These commands are available:

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 71*.

Save to File

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

Right Align 'Value' Column

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

Right Align 'Value' Column

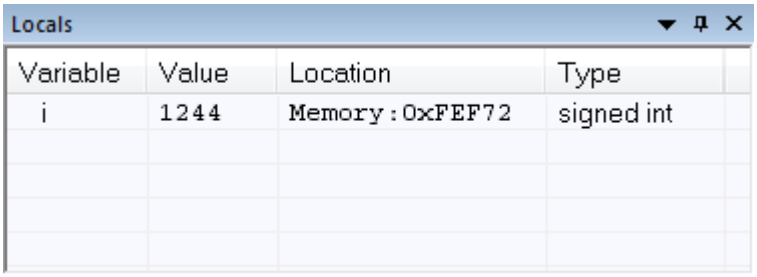
Right-aligns the contents of the **Value** column.

Options

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

Locals window

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



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

See also *Editing in C-SPY windows, page 37*.

Requirements

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

Display area

This area contains these columns:

Expression
The name of the variable. This column is not editable.
A name in square brackets, for example `[fieldname]`, is either the name of a base class for a struct/class object or—if shown in gray and *italic*)—a pointer to the full object when of a class type derived from the original class.

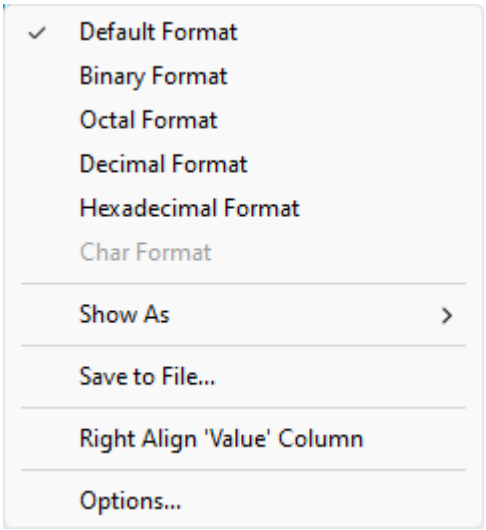
Value
The value of the variable. Values that have changed are highlighted in red.
Dragging text or a variable from another window and dropping it on the **Value** column will assign a new value to the variable in that row.
This column is editable.

Location
The location in memory where this variable is stored.

Type
The data type of the variable.

Context menu

This context menu is available:



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

These commands are available:

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 71*.

Save to File

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

Right Align 'Value' Column

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

Right Align 'Value' Column

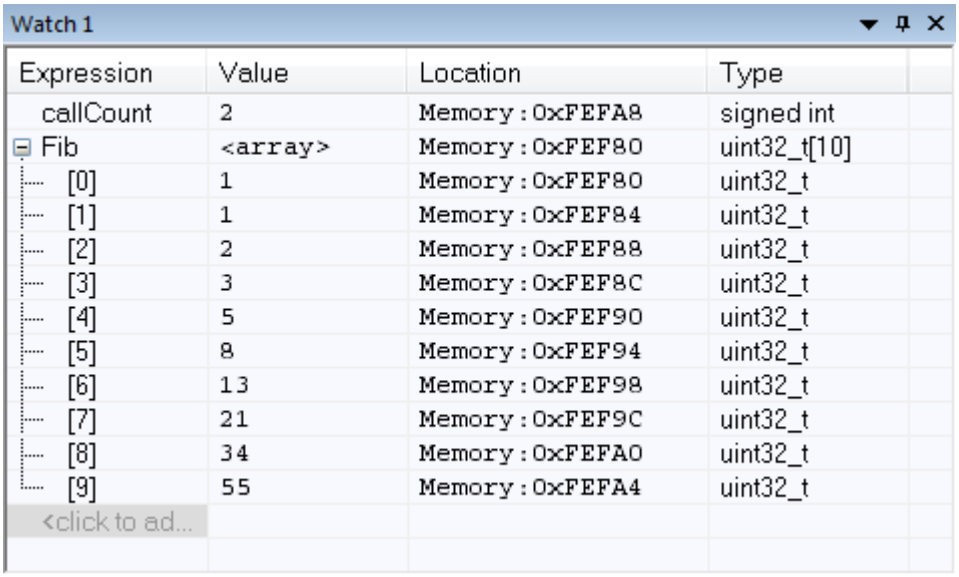
Right-aligns the contents of the **Value** column.

Options

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

Watch window

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



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

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



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

See also *Editing in C-SPY windows, page 37*.

Requirements

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

Display area

This area contains these columns:

Expression — The name of the variable.
A name in square brackets, for example `[fieldname]`, is either the name of a base class for a struct/class object or—if shown in gray and *italic*)—a pointer to the full object when of a class type derived from the original class.

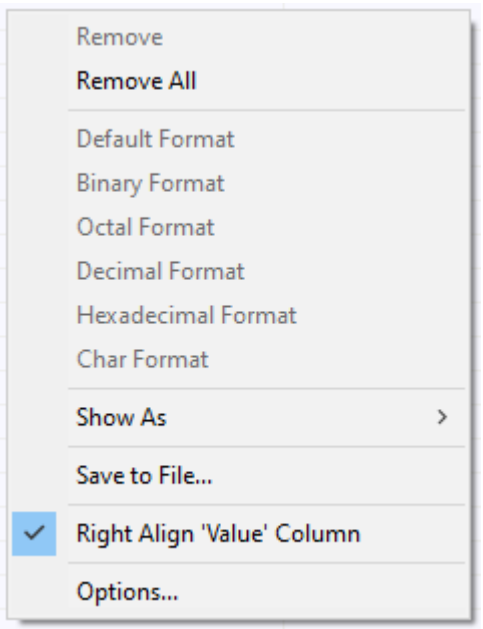
Value — The value of the variable. Values that have changed are highlighted in red.
Dragging text or a variable from another window and dropping it on the **Value** column will assign a new value to the variable in that row.
This column is editable.

Location — The location in memory where this variable is stored.

Type — The data type of the variable.

Context menu

This context menu is available:



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

These commands are available:

Remove — Removes the selected expression from the window.

Remove All — Removes all expressions listed in the window.

Default Format, Binary Format, Octal Format, Decimal Format, Hexadecimal Format, Char Format — Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

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

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

Show As — Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers.For more information, see *Viewing assembler variables, page 71*.

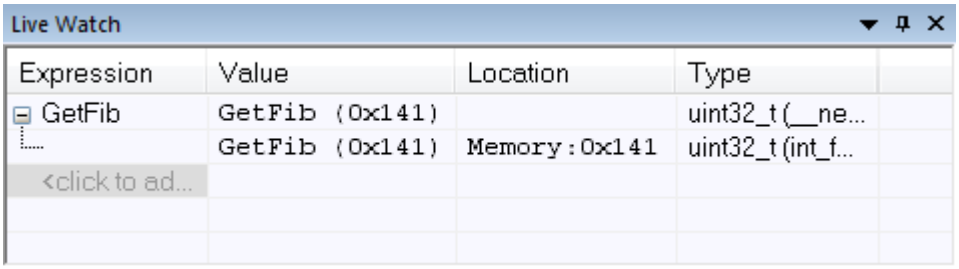
Save to File — Saves content to a file in a tab-separated format.

Right Align 'Value' Column — Right-aligns the contents of the **Value** column.

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

Live Watch window

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



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	Live watch is fully supported for Cortex-M. Access to memory or setting breakpoints is always possible during execution.
Cortex-A/R	Live watch is only possible when you use the C-SPY I-jet driver or J-Link/J-Trace driver, and only when you use the option `--drv_mem_ap`, see *--drv_mem_ap, page 429*.
Armxxx-S	The support for live watch is limited for Armxxx-S cores, see below. Setting hardware breakpoints during execution is always possible for all supported C-SPY drivers.
Arm7/Arm9, including Armxxx-S, and when using the C-SPY J-Link/J-Trace driver	Memory accesses must be made by your application. By adding a small program—a DCC handler—that communicates with the debugger through the DCC unit to your application, memory can be read/written during execution. Software breakpoints can also be set by the DCC handler. Just add the files `JLINKDCC_Process.c` and `JLINKDCC_HandleDataAbort.s` located in `arm\src\debugger\dcc` to your project and call the `JLINKDCC_Process` function regularly, for example every millisecond. In your local copy of the `cstartup` file, modify the interrupt vector table so that data aborts will call the `JLINKDCC_HandleDataAbort` handler. See also *--jlink_dcc_timeout, page 441*.

Table 8. Live watch for the different devices

See also *Editing in C-SPY windows, page 37*.

Requirements

This window is available for all combinations of C-SPY drivers and devices, except for the C-SPY Stellaris driver. However, note that the actual support for live watch is limited for all cores except Cortex-M, see the table above.

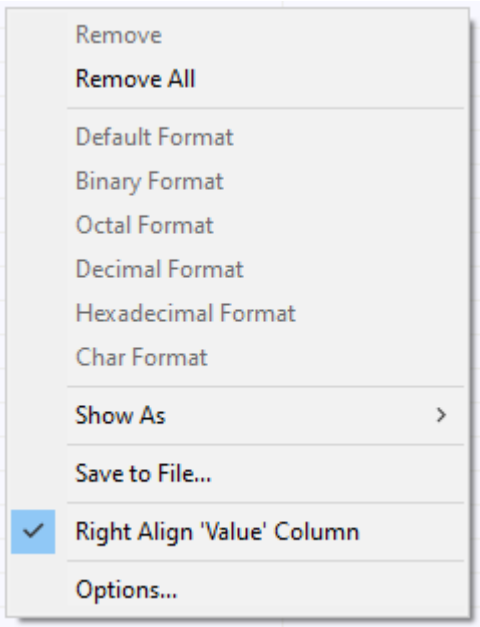
Display area

This area contains these columns:

Expression	The name of the variable. A name in square brackets, for example `[fieldname]`, is either the name of a base class for a struct/class object or—if shown in gray and *italic*)—a pointer to the full object when of a class type derived from the original class.
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:



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

These commands are available:

Remove — Removes the selected expression from the window.

Remove All — Removes all expressions listed in the window.

Default Format, Binary Format, Octal Format, Decimal Format, Hexadecimal Format, Char Format — Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

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

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

Show As — Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers.For more information, see *Viewing assembler variables, page 71*.

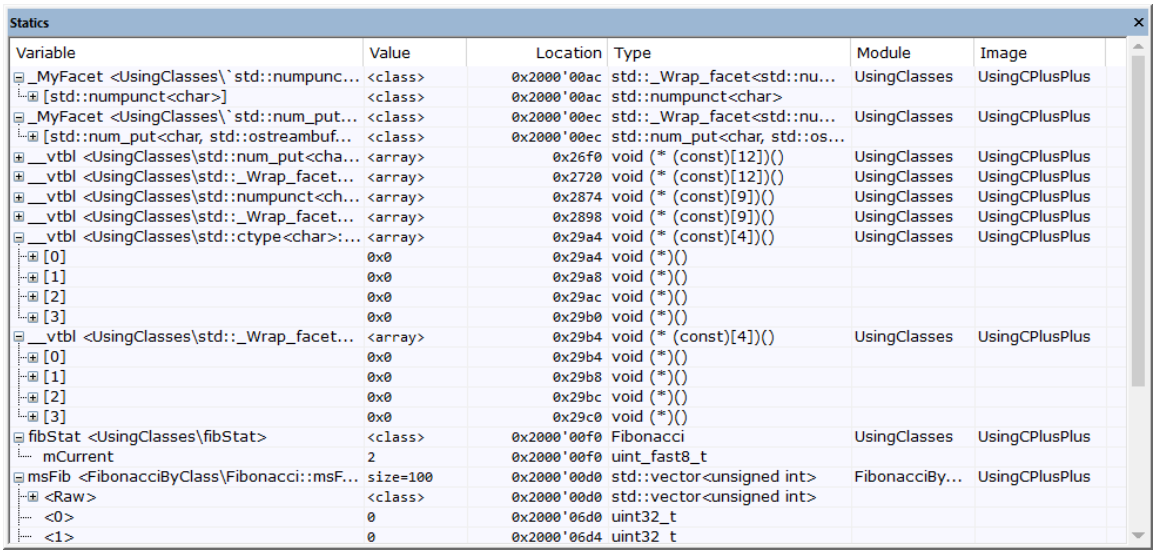
Save to File — Saves content to a file in a tab-separated format.

Right Align 'Value' Column — Right-aligns the contents of the **Value** column.

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

Statics window

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



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

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

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

See also *Editing in C-SPY windows, page 37*.

To select variables to monitor:

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

Requirements

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

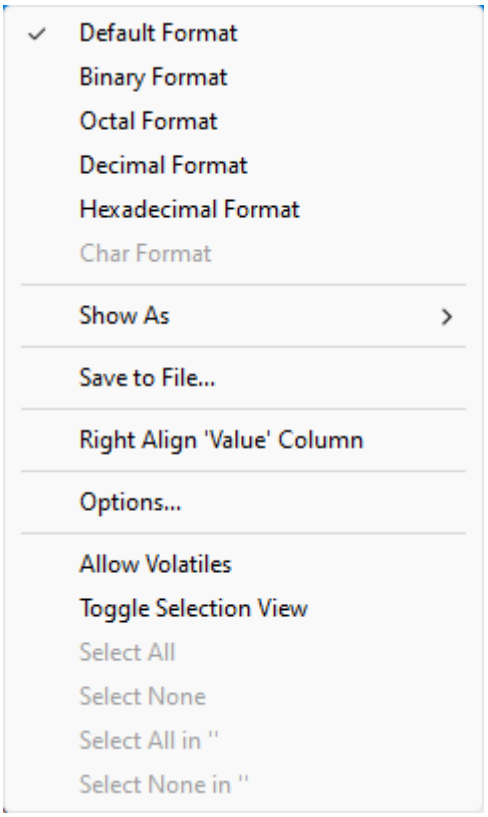
Display area

This area contains these columns:

Variable	The name of the variable. The base name of the variable is followed by the full name, which includes module, class, or function scope. This column is not editable.
Value	The value of the variable. Values that have changed are highlighted in red. Dragging text or a variable from another window and dropping it on the **Value** column will assign a new value to the variable in that row. This column is editable.
Location	The location in memory where this variable is stored.
Type	The data type of the variable.
Module	The module of the variable. A tooltip shows the complete path to the module.
Image	The name of the debug image in which the variable is defined. A tooltip shows the complete path to the image.

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 71*.

Save to File

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

Right Align 'Value' Column

Toggles the left–right alignment of the **Value** column.

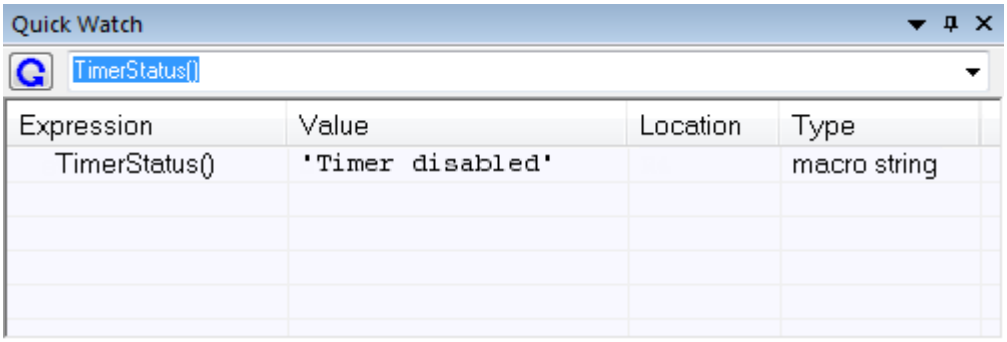
Options

Opens the **IDE Options** dialog box on the **Debugger** category. See the IDE Project Management and Building Guide for Arm for more information.

Allow Volatiles	Toggles the display of volatile variables in the window. When displayed, volatile variables can be recognized by the text in the **Type** column being in ***bold and italic***.
Toggle Selection View	Toggling this command to its selected setting adds a selection box for every displayed variable, that can be used for hiding variables that you are not interested in. By default all variables are selected. The ones you deselect will be hidden when you choose this command again to toggle the command to its deselected setting.
Select All	Selects all variables.
Select None	Deselects all variables.
Select All in *module*	Selects all variables in the selected module.
Select None in *module*	Deselects all variables in the selected module.

Quick Watch window

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



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

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

See also *Editing in C-SPY windows, page 37*.

To evaluate an expression:

1. In the editor window, right-click on the expression you want to examine and choose **Quick Watch** from the context menu that appears.
2. The expression will automatically appear in the **Quick Watch** window.

   Alternatively:
3. In the **Quick Watch** window, type the expression you want to examine in the **Expressions** text box.
4.  Click the **Recalculate** button to calculate the value of the expression.

   For an example, see *Using C-SPY macros, page 332*.

Requirements

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

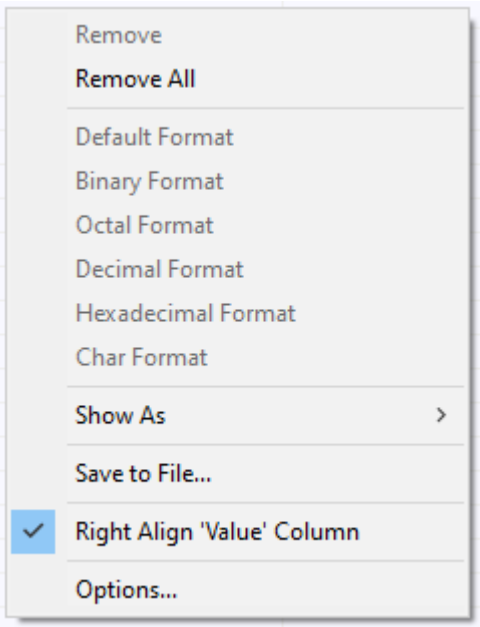
Display area

This area contains these columns:

Expression	The name of the variable. A name in square brackets, for example `[fieldname]`, is either the name of a base class for a struct/class object or—if shown in gray and *italic*)—a pointer to the full object when of a class type derived from the original class.
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:



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

These commands are available:

Remove — Removes the selected expression from the window.

Remove All — Removes all expressions listed in the window.

Default Format, Binary Format, Octal Format, Decimal Format, Hexadecimal Format, Char Format — Changes the display format of expressions. The display format setting affects different types of expressions in different ways. Your selection of display format is saved between debug sessions. These commands are available if a selected line in the window contains a variable.

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

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

Show As — Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers.For more information, see *Viewing assembler variables, page 71*.

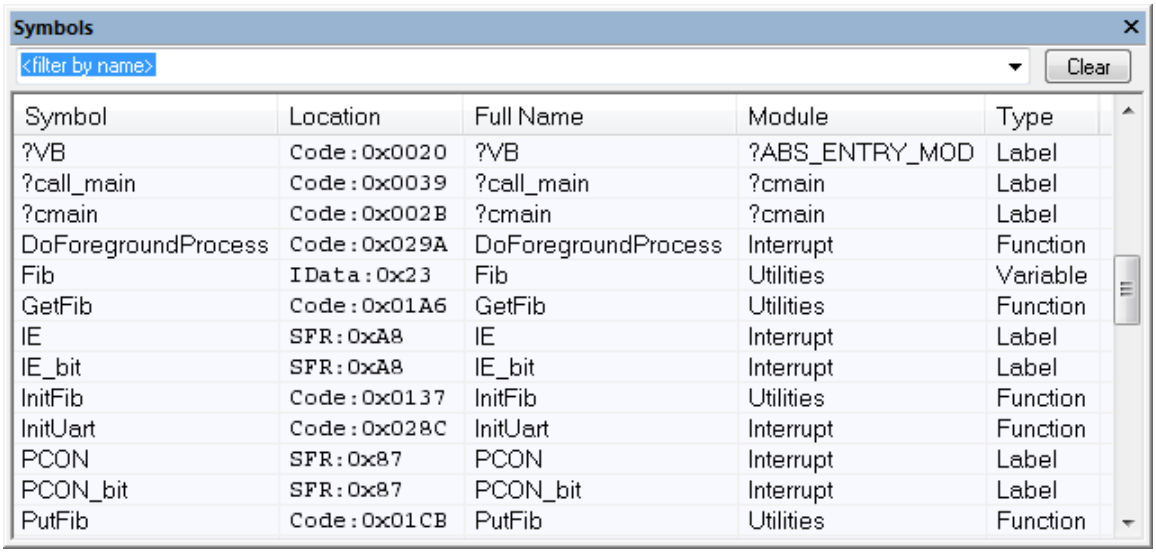
Save to File — Saves content to a file in a tab-separated format.

Right Align 'Value' Column — Right-aligns the contents of the **Value** column.

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

Symbols window

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



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

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

Requirements

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

Toolbar

The toolbar contains:

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

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

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

Display area

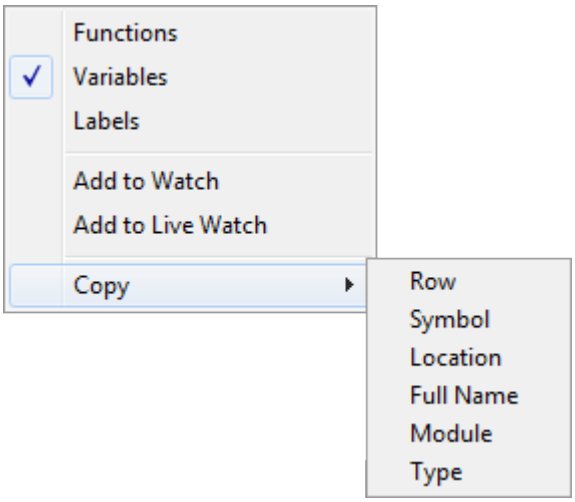
This area contains these columns:

Symbol	The symbol name.
Location	The memory address.
Full name	The symbol name—often the same as the contents of the **Symbol** column but differs for example for C++ member functions.
Module	The program module where the symbol is defined.
Type	The symbol type, whether it is a function, label, or variable.

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

Context menu

This context menu is available:



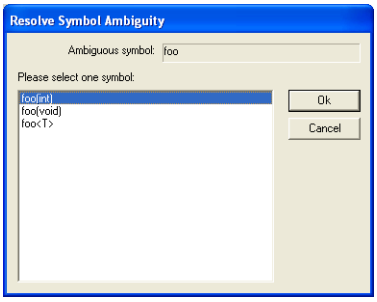
These commands are available:

Functions	Toggles the display of function symbols on or off in the list.
Variables	Toggles the display of variables on or off in the list.
Labels	Toggles the display of labels on or off in the list.
Add to Watch	Adds the selected symbol to the **Watch** window.
Add to Live Watch	Adds the selected symbol to the **Live Watch** window.
Copy	Copies the contents of the cells on the selected line.

Row	Copies all contents of the selected line.
Symbol	Copies the contents of the **Symbol** cell on the selected line.
Location	Copies the contents of the **Location** cell on the selected line.
Full Name	Copies the contents of the **Full Name** cell on the selected line.
Module	Copies the contents of the **Module** cell on the selected line.
Type	Copies the contents of the **Type** cell on the selected line.

Resolve Symbol Ambiguity dialog box

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



Requirements

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

Ambiguous symbol

Indicates which symbol that is ambiguous.

Please select one symbol

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

Breakpoints

Contents

Introduction to setting and using breakpoints 90
Reasons for using breakpoints 90
Briefly about setting breakpoints 91
Breakpoint types 91
Breakpoint icons 93
Breakpoints in the C-SPY simulator 93
Breakpoints in the C-SPY hardware debugger drivers 93
Breakpoint consumers 93
Breakpoint options 94
Setting breakpoints 94
Various ways to set a breakpoint 94
Toggling a simple code breakpoint 95
Setting breakpoints using the dialog box 95
Setting a data breakpoint in the Memory window 96
Setting breakpoints using system macros 96
Setting a breakpoint on an exception vector 97
Setting breakpoints in __ramfunc declared functions 98
Useful breakpoint hints 98
Reference information on breakpoints 99
Breakpoints window 99
Breakpoint Usage window 101
Code breakpoints dialog box 102
JTAG Watchpoints dialog box 103
Log breakpoints dialog box 106
Data breakpoints dialog box 107
Data Log breakpoints dialog box 110
Data Log breakpoints dialog box (C-SPY hardware drivers) 111
Breakpoints options 112
Immediate breakpoints dialog box 114
Vector Catch dialog box 115
Flash breakpoints dialog box 116
Enter Location dialog box 117
Resolve Source Ambiguity dialog box 118

INTRODUCTION TO SETTING AND USING BREAKPOINTS

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 93*.

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 50*.

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.



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

Breakpoint types

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

Code breakpoints

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

Log breakpoints

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

Trace Start/Stop Trigger breakpoints

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

Data breakpoints

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

The execution will usually stop directly after the instruction that accessed the data has been executed. However, on some newer Arm devices like Cortex-M55, data breakpoints might not stop the execution until several instructions later. Moreover, if the CPU is halted for any reason, any such “delayed” data breakpoint will never have any effect. Examples: If two consecutive instructions read from two different locations that both have a data breakpoint, the second one will not have any effect. Likewise, if a data breakpoint is immediately followed by a code breakpoint, then that data breakpoint will not have any effect.

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 167*.

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

For some cores and devices, 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 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 Arm Limited:

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

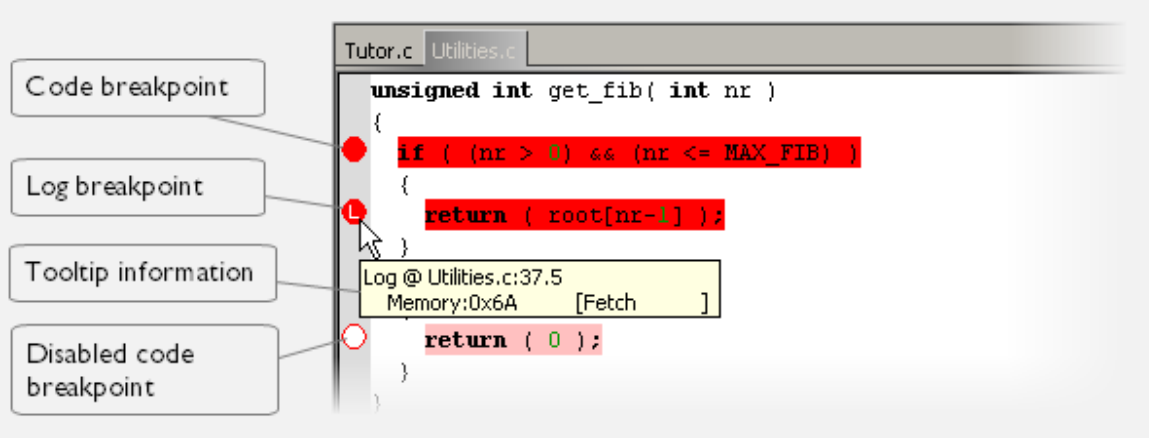
Flash breakpoints

Software code breakpoints (breakpoints that rely on writing breakpoint instructions into memory) can be set in flash memory or other non-volatile executable memory using the flash breakpoint mechanism. This involves programming of the flash memory using a flash loader in the same way as is done when downloading an application.

This breakpoint type is available for the C-SPY I-jet driver. Other debug probes might offer the possibility to set breakpoints in flash memory using other mechanisms. J-Link, for example, is one of these.

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.

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

Breakpoints in the C-SPY simulator

For some cores and devices, the C-SPY simulator supports all breakpoint types. The C-SPY simulator always supports code, data, and log breakpoints for all cores and devices. The number of breakpoints is unlimited.

Breakpoints in the C-SPY hardware debugger drivers

Using the C-SPY drivers for hardware debugger systems, you can set various breakpoint types. The amount of breakpoints you can set depends on the number of *hardware breakpoints* available on the target system or whether 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.

Breakpoint 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
- J-Link/J-Trace
- CMSIS-DAP
- ST-LINK
- TI XDS

For more information, see *Breakpoints options, page 112*.

SETTING BREAKPOINTS

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.

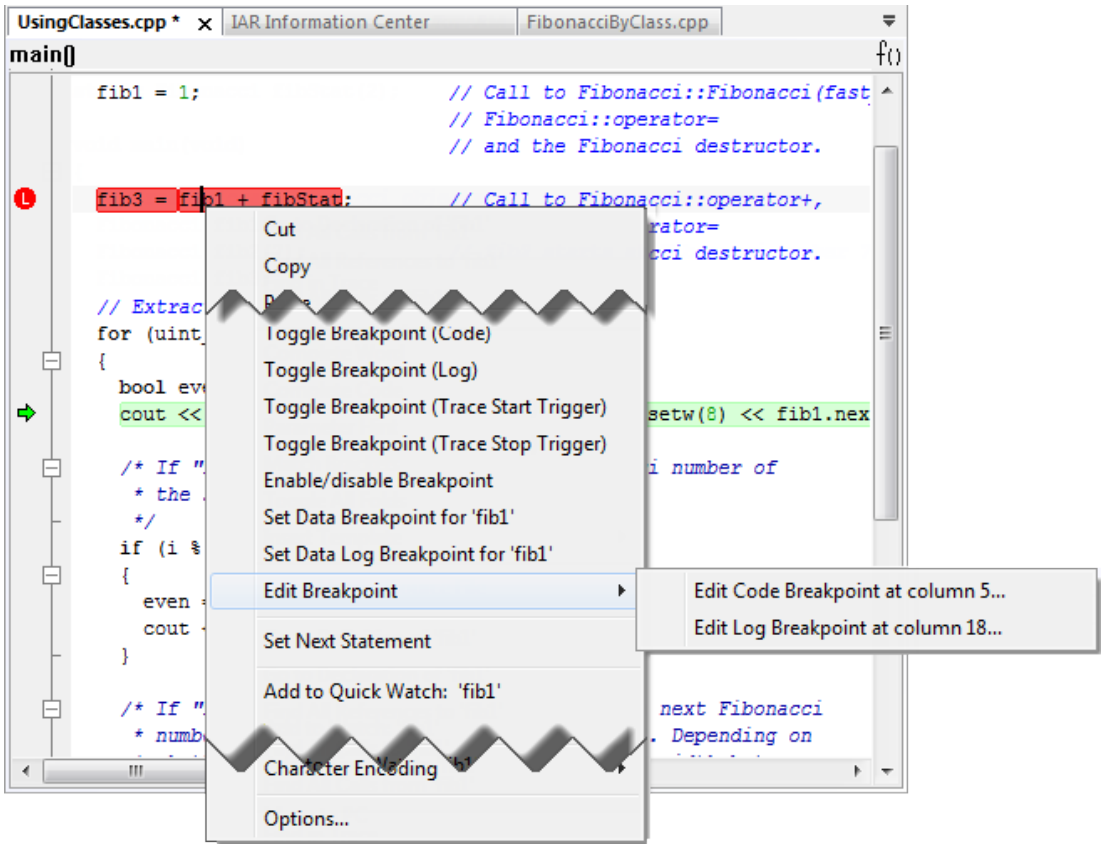
You can open the dialog box from the context menu available in the editor window, **Breakpoints** window, and in the **Disassembly** window.

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.

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.



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.

C-SPY macro for breakpoints	Simulator	I-jet	J-Link/J-Trace	CMSIS-DAP	CADI	E2
`__setCodeBreak`	Yes	Yes	Yes	Yes	Yes	Yes
`__setDataBreak`	Yes	Yes	—	Yes	—	—
`__setLogBreak`	Yes	Yes	Yes	Yes	—	Yes
`__setDataLogBreak`	Yes[1]	Yes	—	—	—	—
`__setSimBreak`	Yes[1]	—	—	—	—	—
`__setTraceStartBreak`	Yes[1]	Yes	—	—	—	—
`__setTraceStopBreak`	Yes[1]	Yes	—	—	—	—
`__clearBreak`	Yes[1]	Yes	Yes	Yes	Yes	Yes

Table 9. C-SPY macros for breakpoints

1. Not for all cores and devices.

C-SPY macro for breakpoints	GDB Server	ST-LINK and TI MSP-FET	PE micro	TI Stellaris / TI XDS	Nu-Link	G+Link
`__setCodeBreak`	Yes	Yes	Yes	Yes	Yes	Yes
`__setDataBreak`	—	—	—	—	—	Yes
`__setLogBreak`	Yes	Yes	Yes	Yes	Yes	Yes
`__setDataLogBreak`	—	—	—	—	—	—
`__setSimBreak`	—	—	—	—	—	—
`__setTraceStartBreak`	—	—	—	—	—	—
`__setTraceStopBreak`	—	—	—	—	—	—
`__clearBreak`	Yes	Yes	Yes	Yes	Yes	Yes

Table 10. C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros, page 343*.

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 332*.

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 115*.

For the C-SPY I-jet driver and the C-SPY J-Link/J-Trace driver, it is also possible to set breakpoints directly on a vector already in the options dialog box, see *J-Link/J-Trace — Setup, page 490*.

This procedure applies to the C-SPY I-jet driver and the C-SPY J-Link/J-Trace 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, page 112*.
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, page 112*, 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 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.

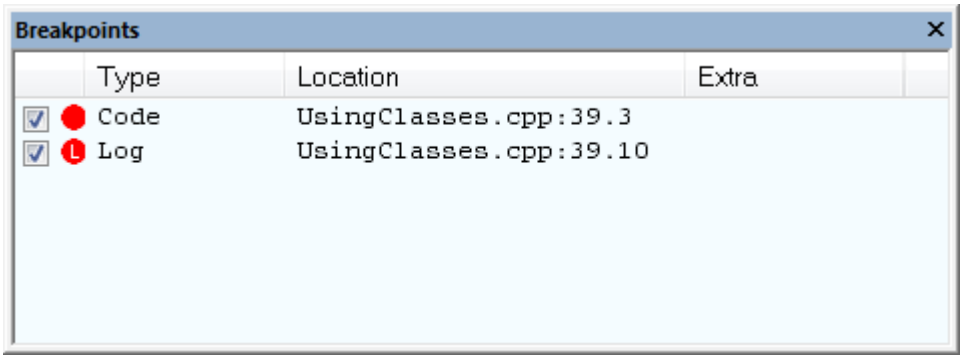
REFERENCE INFORMATION ON BREAKPOINTS

See also:

- *Reference information on C-SPY system macros, page 343*
- *Reference information on trace, page 161*

Breakpoints window

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



This window lists all breakpoints you define.

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

Requirements

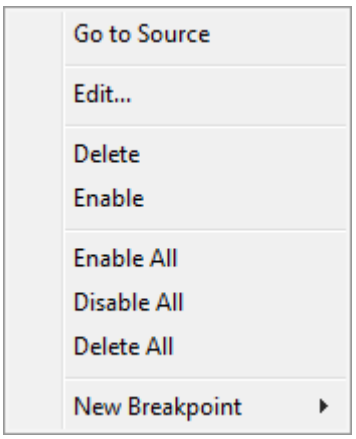
Can be used with all C-SPY debugger drivers and debug probes.

Display area

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

Context menu

This context menu is available:

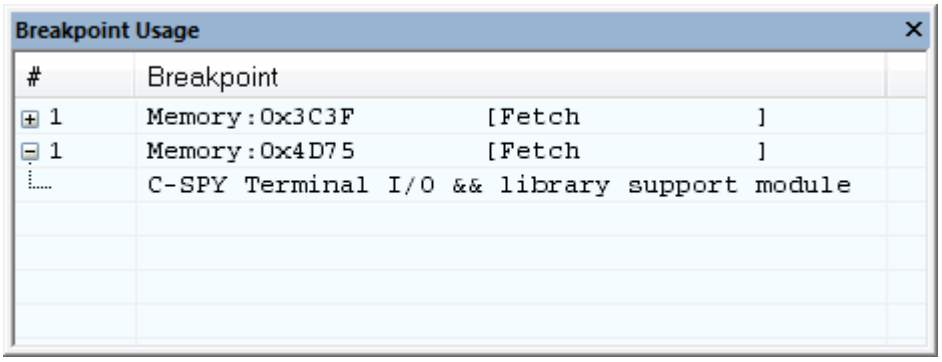


These commands are available:

Go to Source	Moves the insertion point to the location of the breakpoint, if the breakpoint has a source location. Double-click a breakpoint in the **Breakpoints** window to perform the same command.
Edit	Opens the breakpoint dialog box for the breakpoint you selected.
Delete	Deletes the breakpoint. Press the Delete key to perform the same command.
Enable	Enables the breakpoint. The check box at the beginning of the line will be selected. You can also perform the command by manually selecting the check box. This command is only available if the breakpoint is disabled.
Disable	Disables the breakpoint. The check box at the beginning of the line will be deselected. You can also perform this command by manually deselecting the check box. This command is only available if the breakpoint is enabled.
Enable All	Enables all defined breakpoints.
Disable All	Disables all defined breakpoints.
Delete All	Deletes all defined breakpoints.
New Breakpoint	Displays a submenu where you can open the breakpoint dialog box for the available breakpoint types. All breakpoints you define using this dialog box are preserved between debug sessions.

Breakpoint Usage window

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



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 93*.

Requirements

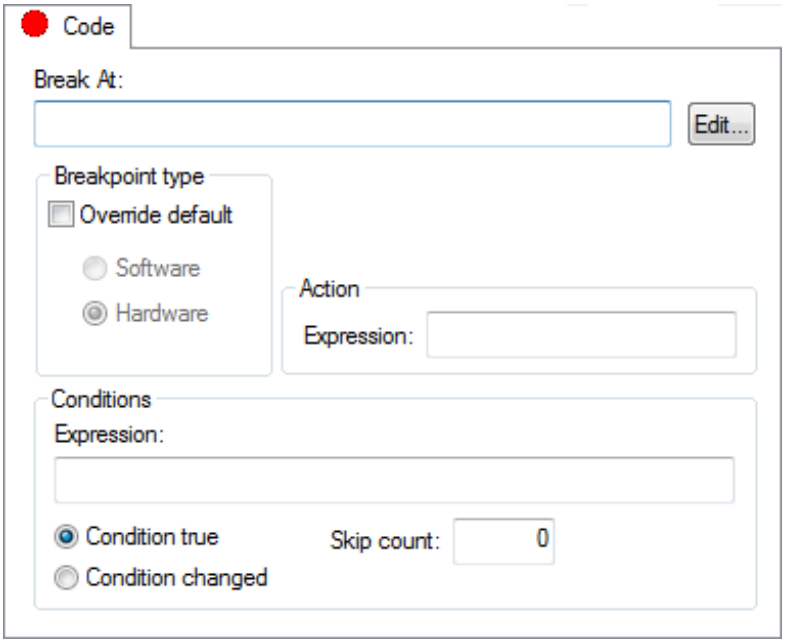
Can be used with all C-SPY debugger drivers and debug probes.

Display area

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

Code breakpoints dialog box

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



This figure reflects the C-SPY simulator.

Use the **Code** breakpoints dialog box to set a code breakpoint, see *Setting breakpoints using the dialog box, page 95*.

Requirements

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

Break At

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box, page 117*.

Breakpoint type

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

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

- The C-SPY I-jet driver
- The C-SPY CMSIS-DAP driver
- The C-SPY GDB Server driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY TI MSP-FET driver
- The C-SPY TI XDS 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.



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

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints, page 98*.

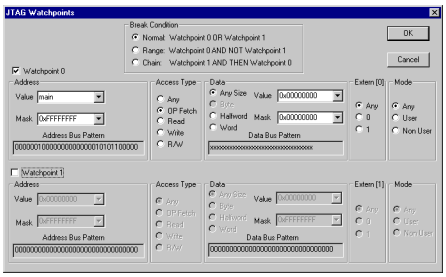
Conditions

Specify simple or complex conditions:

Expression	Specify a valid C-SPY expression, see *C-SPY expressions, page 68*.
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

The J-Link/J-Trace driver and a supported device.

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.

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 *C-SPY expressions, page 68*.
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 *C-SPY expressions, page 68*.
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.

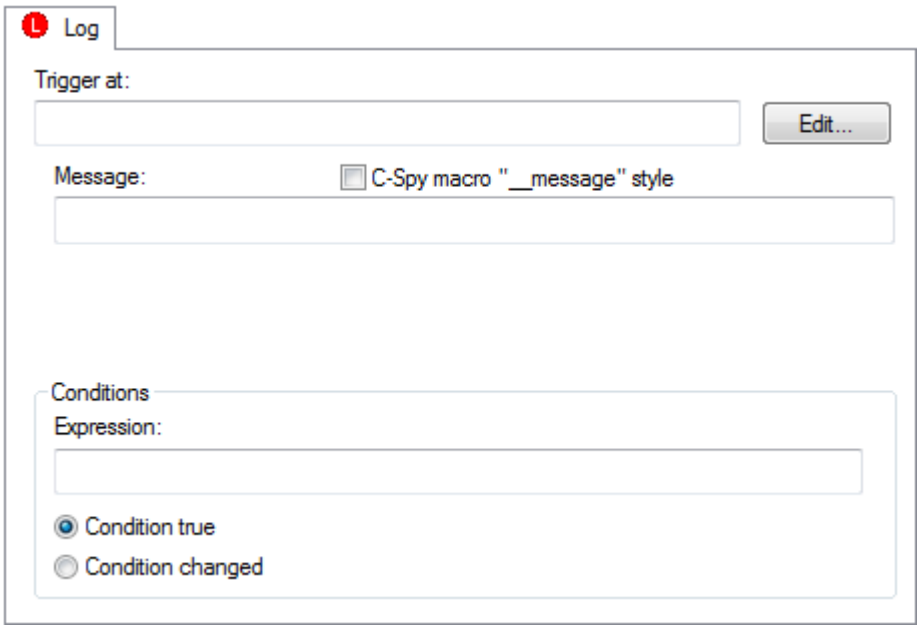
Mode

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.

Log breakpoints dialog box

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



This figure reflects the C-SPY simulator.

Use the **Log** breakpoints dialog box to set a log breakpoint, see *Setting breakpoints using the dialog box, page 95*.

Requirements

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

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box, page 117*.

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 337*.

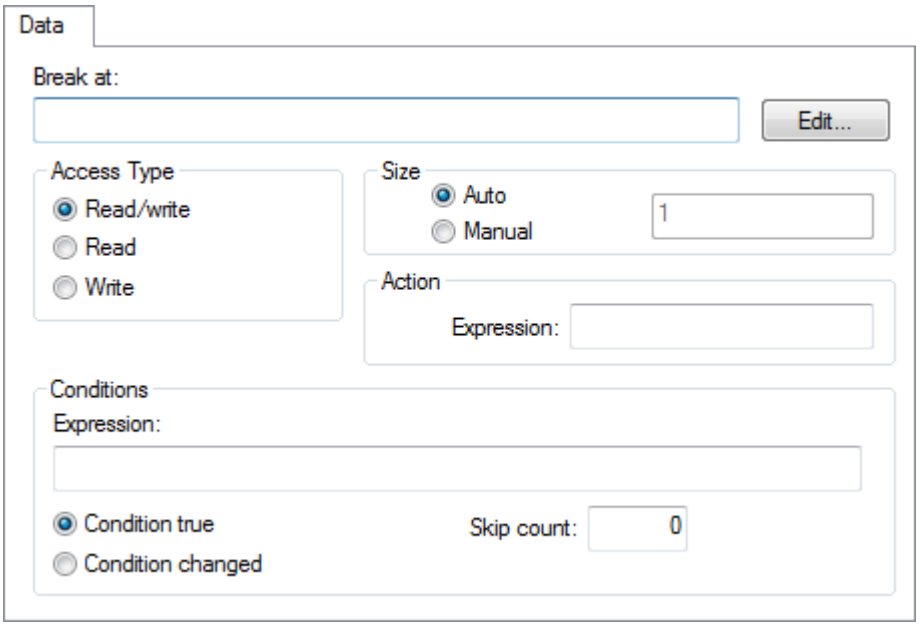
Conditions

Specify simple or complex conditions:

Expression	Specify a valid C-SPY expression, see *C-SPY expressions, page 68*.
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, see *Setting breakpoints using the dialog box, page 95*. 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 CMSIS-DAP driver
- The C-SPY GDB Server driver
- The C-SPY G+LINK driver
- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver
- The C-SPY TI MSP-FET driver
- The C-SPY TI Stellaris driver
- The C-SPY TI XDS driver and a Cortex-M device

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 117*.

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 98*.

Conditions

Specify simple or complex conditions:

Expression	Specify a valid C-SPY expression, see *C-SPY expressions, page 68*.
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.

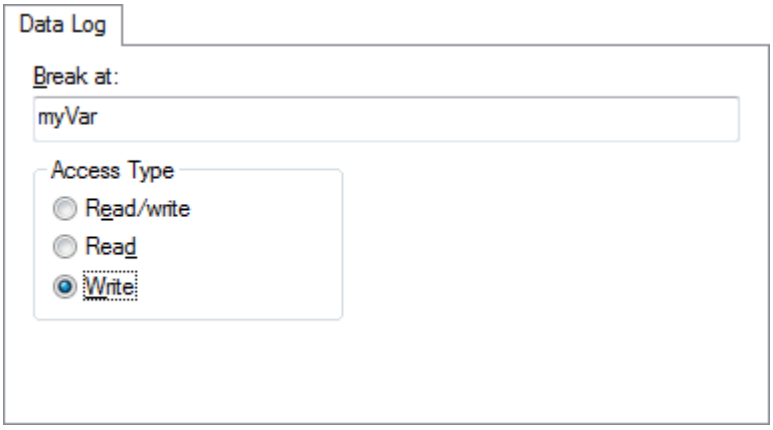
The **Match data** options are only available for I-jet, I-jet Trace, J-Link/J-Trace and ST-LINK, and when using an Arm7/9 or a Cortex-M device.

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

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 *Setting breakpoints using the dialog box, page 95*.

See also *Data Log breakpoints, page 92* and *Getting started using data logging, page 198*.

Requirements

The C-SPY simulator.

Break At

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

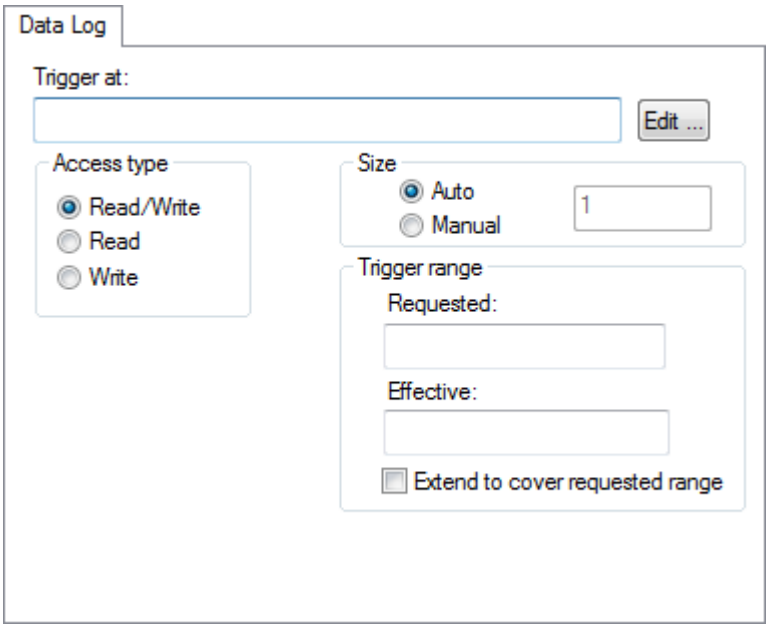
Access Type

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 92* and *Getting started using data logging, page 198*.

Requirements

A device and a debug probe with support for SWO.

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 117*.

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.

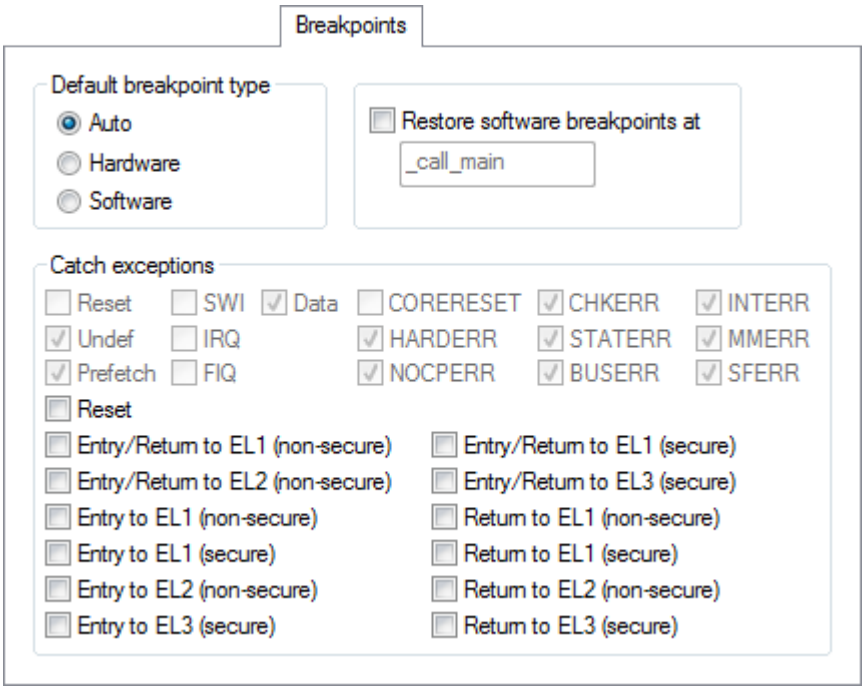
Trigger range

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

The **Breakpoints** options 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 driver
- The C-SPY J-Link/J-Trace 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

Restores software breakpoints that were overwritten during system startup.

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

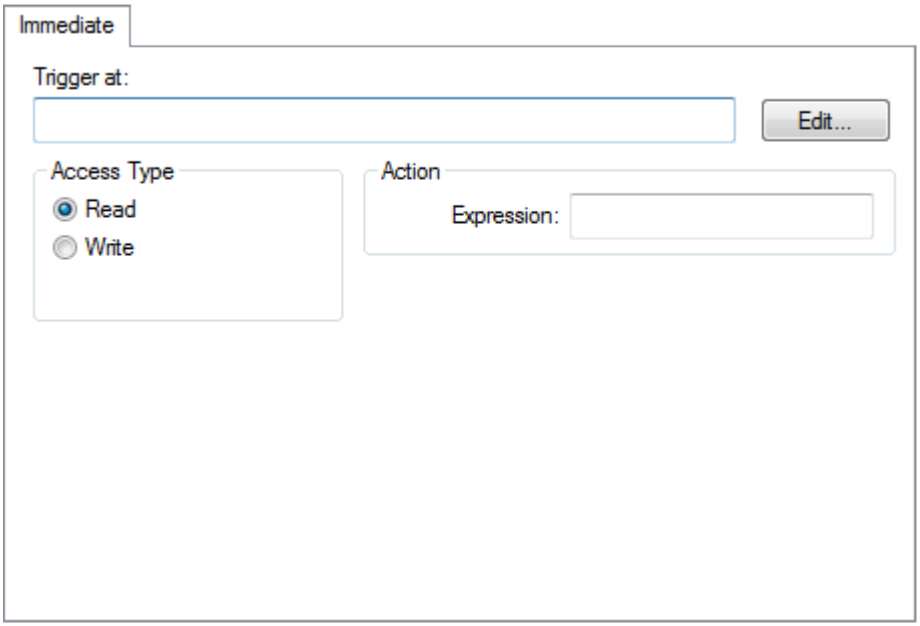
Sets a breakpoint directly on a vector in the interrupt vector table, without using a hardware breakpoint. This option is available for Arm9/10/11 and all Cortex devices. The settings you make will work as default settings for the project. However, you can override these default settings during the debug session by using the **Vector Catch** dialog box, see *Setting a breakpoint on an exception vector, page 97*.

The settings you make will be preserved during debug sessions.

This option is supported by the C-SPY I-jet 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, you can use the **Immediate** breakpoints dialog box to set an immediate breakpoint, see *Setting breakpoints using the dialog box, page 95*. Immediate breakpoints do not stop execution at all—they only suspend it temporarily.

Immediate breakpoints are not available for all cores and devices.

Requirements

The C-SPY simulator. Not available for all cores and devices.

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 117*.

Access Type

Selects the type of memory access that triggers the breakpoint:

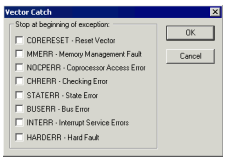
Read	Reads from location.
Write	Writes to location.

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 98*.

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 Arm9/10/11 and all Cortex 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.

For the C-SPY I-jet driver and the C-SPY J-Link/J-Trace driver, it is also possible to set breakpoints directly on a vector already in the options dialog box, see *J-Link/J-Trace — Setup, page 490*.

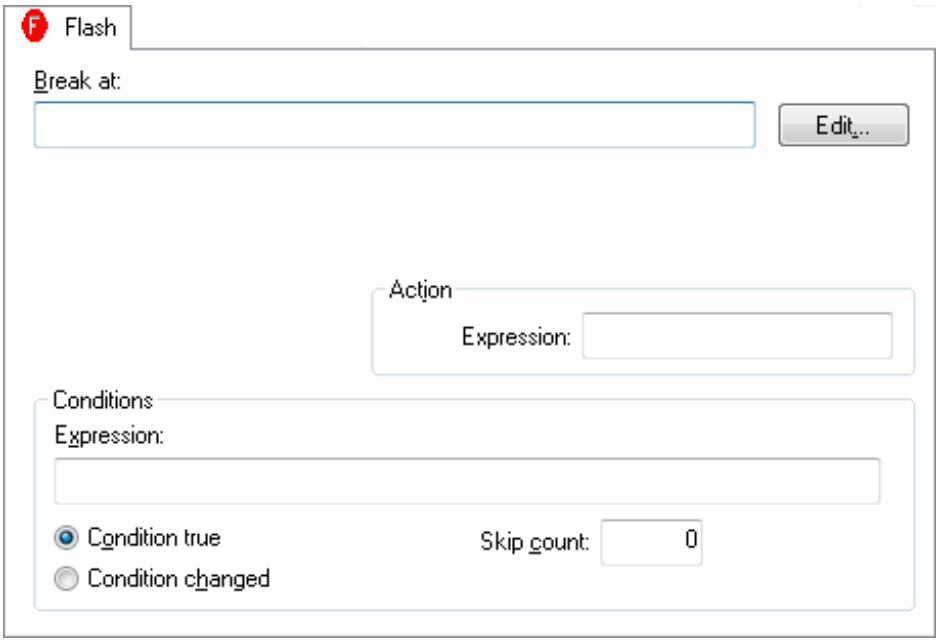
Requirements

One of these alternatives:

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

Flash breakpoints dialog box

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



Use the **Flash** breakpoints dialog box to set a flash breakpoint.

Requirements

The C-SPY I-jet driver.

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 117*.

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 98*.

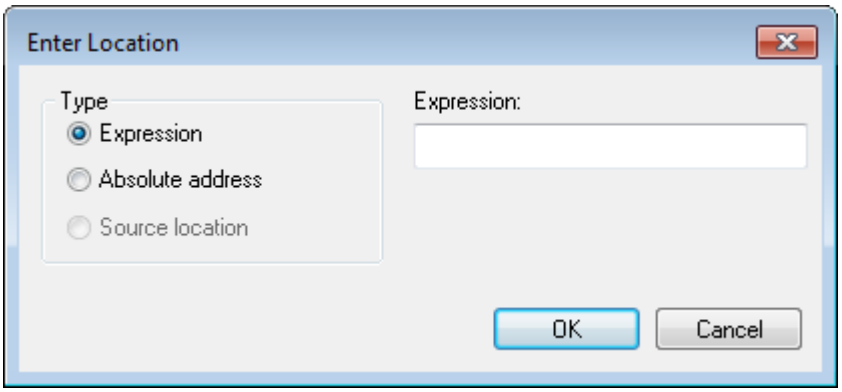
Conditions

Specify simple or complex conditions:

Expression	Specify a valid C-SPY expression, see *C-SPY expressions, page 68*.
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.

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.

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 68*.

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 121*.

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.

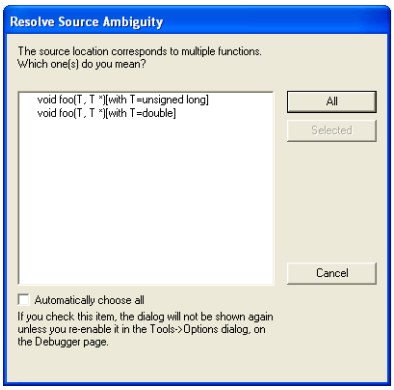
`column` specifies the column in which you want the breakpoint.

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

Contents

Introduction to monitoring memory and registers 120
Briefly about monitoring memory and registers 120
C-SPY memory zones 121
Memory configuration for the C-SPY simulator 122
Memory configuration for C-SPY hardware debugger drivers 122
Monitoring memory and registers 123
Defining application-specific register groups 123
Monitoring stack usage 123
Reference information on memory and registers 126
Memory window 126
Memory Save dialog box 129
Memory Restore dialog box 130
Fill dialog box 131
Symbolic Memory window 132
Stack window 135
Registers window 138
Register User Groups Setup window 141
SFR Setup window 142
Edit SFR dialog box 145
Memory Configuration dialog box for the C-SPY simulator 146
Edit Memory Range dialog box for the C-SPY simulator 148
Memory Configuration dialog box for C-SPY hardware debugger drivers 149
Edit Memory Range dialog box for C-SPY hardware debugger drivers 151

INTRODUCTION TO MONITORING MEMORY AND REGISTERS

Briefly about monitoring memory and registers

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

- The **Memory** window

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

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

  Displays the contents of the stack, including how stack variables are laid out in memory. In addition, integrity checks of the stack can be performed to detect and warn about problems with stack overflow.

For example, the **Stack** window is useful for determining the optimal size of the stack. You can open up to two instances of this window, each showing different stacks or different display modes of the same stack.

- The **Register**s 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 **Registers** window. Instead you can divide registers into *application-specific groups*. You can choose to load either predefined register groups or define your own groups. You can open several instances of this window, each showing a different register group.

- The **SFR Setup** window

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

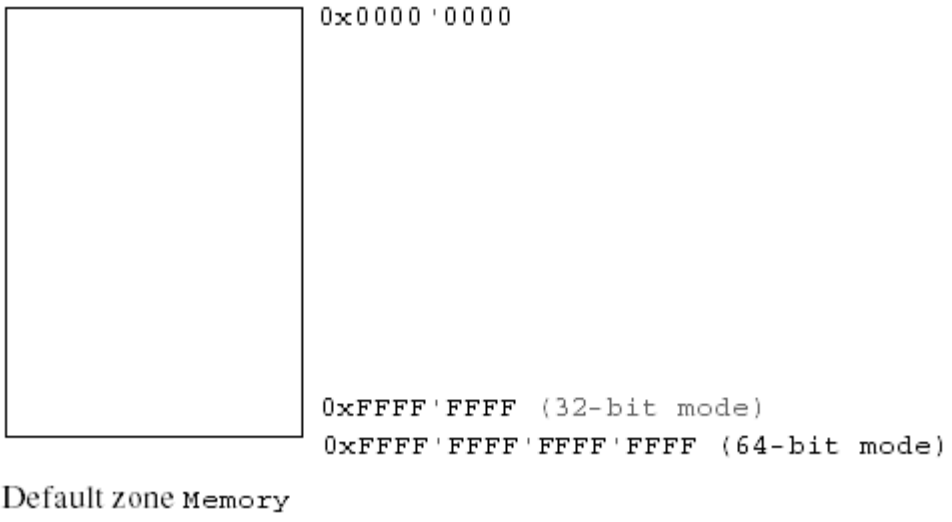
To view the memory contents for a specific variable, simply drag the variable to the **Memory** window or the **Symbolic Memory** window. The memory area where the variable is located will appear.



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

C-SPY memory zones

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



For the C-SPY I-jet driver, you can specify the automatic selection of access width in the **Edit Memory Range** dialog box, see *Edit Memory Range dialog box for C-SPY hardware debugger drivers, page 151*.

Memory configuration for the C-SPY simulator

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

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

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

For more information, see *Memory Configuration dialog box for the C-SPY simulator, page 146*.

Memory configuration for C-SPY hardware debugger drivers

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

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

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

  Note that if you specify the cache type **ROM/Flash**, C-SPY treats such memory as constant during the whole debug session (which improves efficiency, when updating some C-SPY windows). If your application modifies flash memory during runtime, do not use the **ROM/Flash** cache type.
- You can prevent C-SPY from accessing memory outside specified memory address ranges, which can be important for certain hardware.

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

For more information, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.

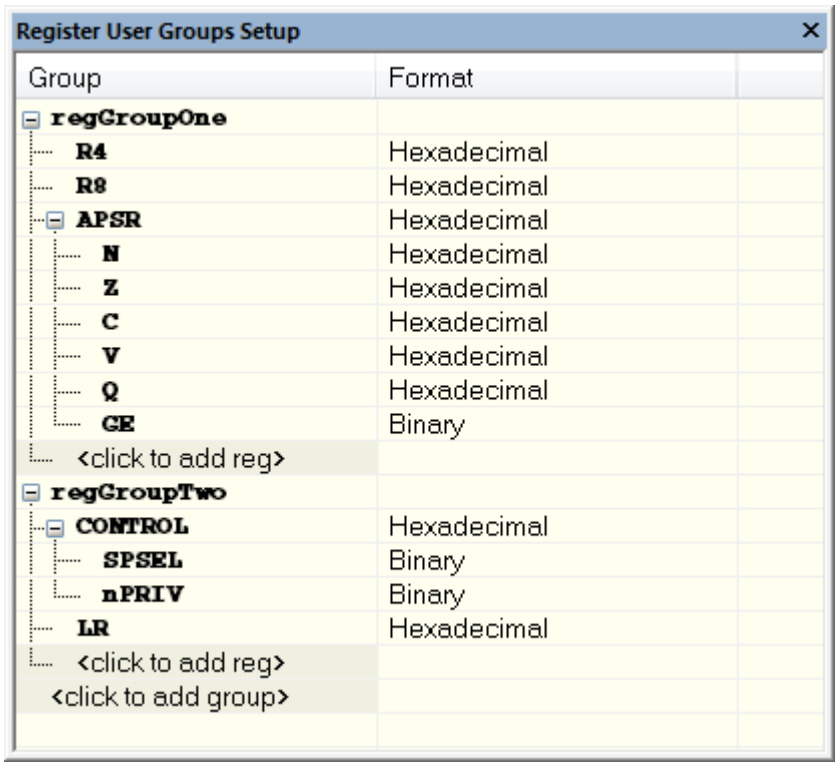
MONITORING MEMORY AND REGISTERS

Defining application-specific register groups

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

To define application-specific register groups:

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



   Right-clicking in the window displays a context menu with commands. For information about these commands, see *Register User Groups Setup window, page 141*.
2. Click on `<click to add group>` and specify the name of your group, for example **My Timer Group** and press Enter.
3. Underneath the group name, click on `<click to add reg>` and type the name of a register, and press Enter. You can also drag a register name from another window in the IDE. Repeat this for all registers that you want to add to your group.
4. As an optional step, right-click any registers for which you want to change the integer base, and choose **Format** from the context menu to select a suitable base.
5. When you are done, your new group is now available in the **Registers** windows.

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

   If a certain SFR that you need cannot be added to a group, you can register your own SFRs. For more information, see *SFR Setup window, page 142*.

Monitoring stack usage

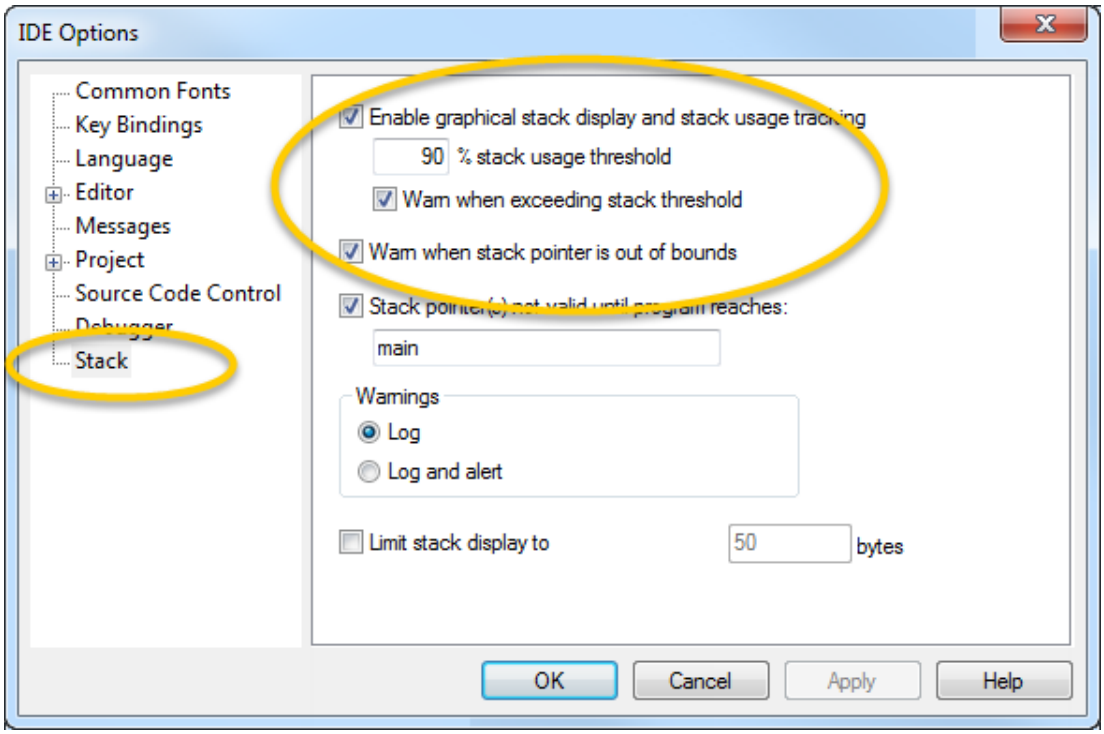
These are the two main use cases for the **Stack** window:

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

In both cases, C-SPY retrieves information about the defined stack size and its allocation from the definition in the linker configuration file of the section holding the stack. If you, for some reason, have modified the stack initialization in the system startup code, `cstartup`, you should also change the section definition in the linker configuration file accordingly, otherwise the **Stack** window cannot track the stack usage. For more information, see the *IAR C/C++ Development Guide for Arm*.

To monitor stack memory usage:

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



2. Start C-SPY.

   When your application is first loaded, and upon each reset, the memory for the stack area is filled with the dedicated byte value `0xCD` before the application starts executing.

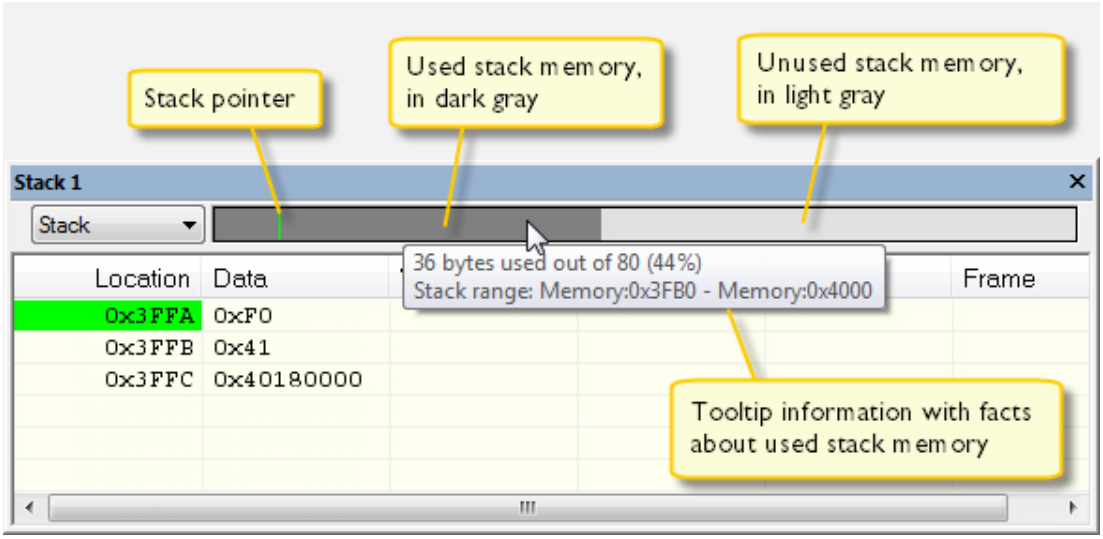
3. Choose **View>Stack>Stack 1** to open the **Stack** window.

   Note that you can open up to two **Stack** windows, each showing a different stack—if several stacks are available—or the same stack with different display settings.

4. Start executing your application.

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

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



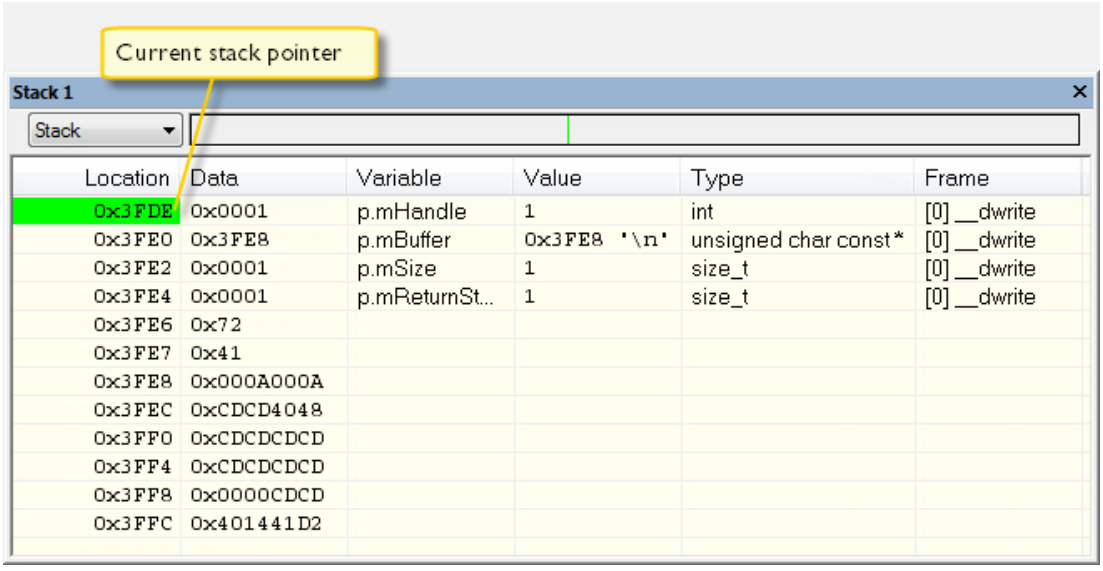
Although this is a reasonably reliable way to track stack usage, there is no guarantee that a stack overflow is detected. For example, a stack *can* incorrectly grow outside its bounds, and even modify memory outside the stack area, without actually modifying any of the bytes near the end of the stack range. Likewise, your application might modify memory within the stack area by mistake.

To monitor the stack memory content:

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

   Note that you can access various context menus in the display area from where you can change display format, etc.
4. Start executing your application.

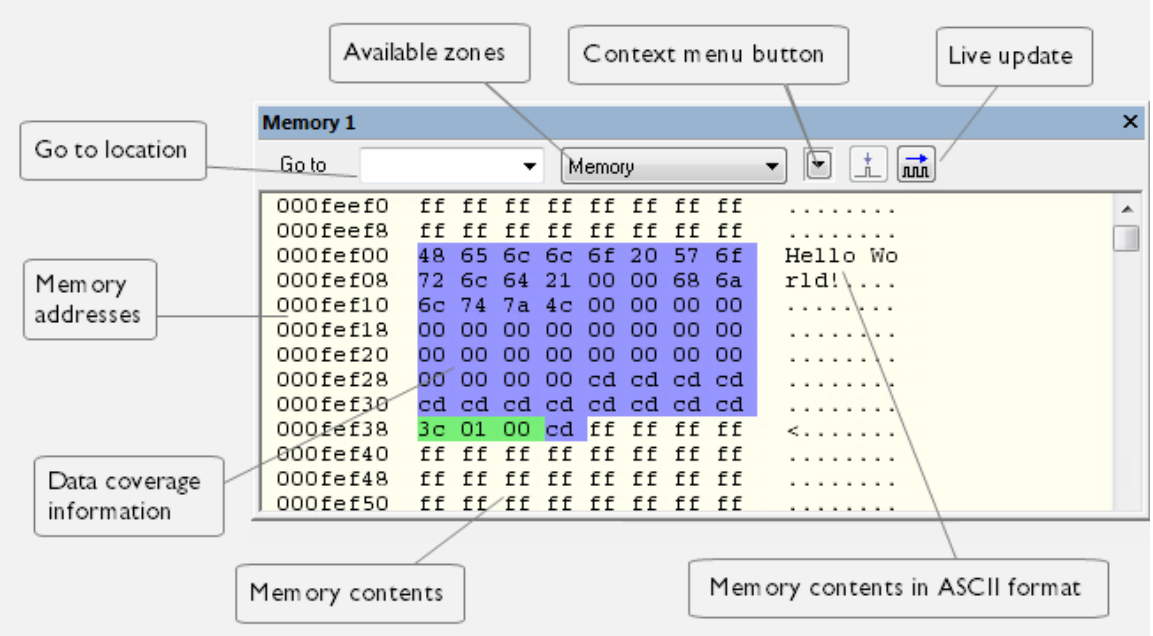
   Whenever execution stops, you can monitor the stack memory, for example to see function parameters that are passed on the stack:



REFERENCE INFORMATION ON MEMORY AND REGISTERS

Memory window

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



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



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

See also *Editing in C-SPY windows, page 37*.

Requirements

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

Toolbar

The toolbar contains:

Go to	The memory location or symbol you want to view.
Zone	Selects a memory zone, see *C-SPY memory zones, page 121*.
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:

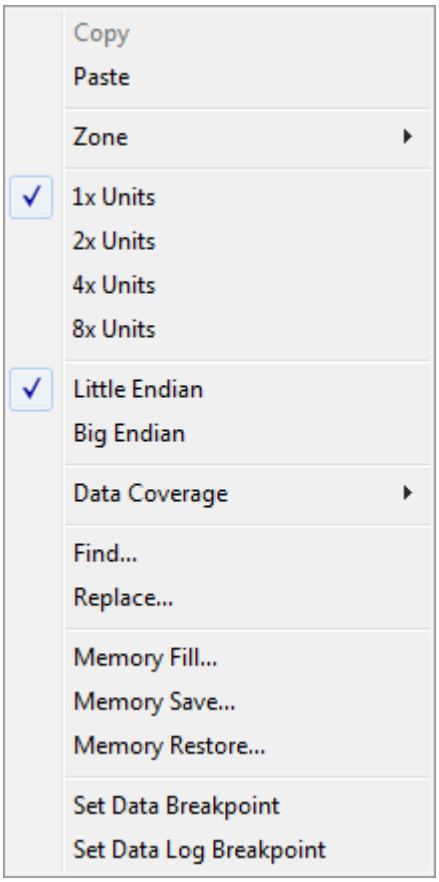
Yellow	Indicates data that has been read.
Blue	Indicates data that has been written
Green	Indicates data that has been both read and written.



Data coverage is not supported by all C-SPY drivers. Data coverage is supported by the C-SPY simulator for many cores and devices.

Context menu

This context menu is available:



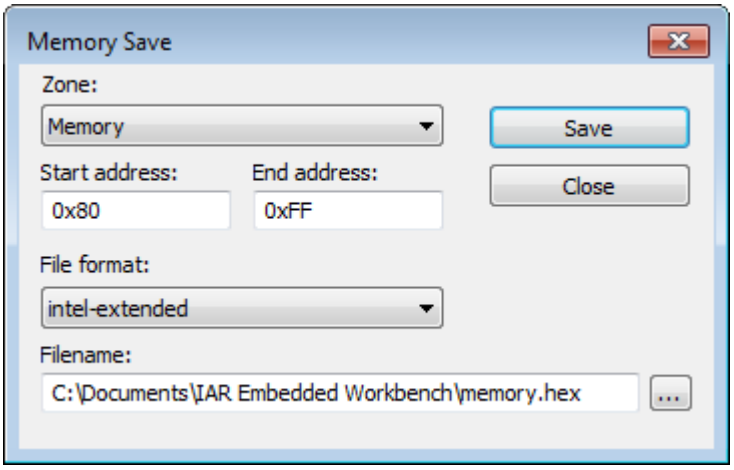
These commands are available:

Copy, Paste	Standard editing commands.
Zone	Selects a memory zone, see *C-SPY memory zones, page 121*.
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 131*.
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 129*.
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 130*.
Set Data Breakpoint	Sets breakpoints directly in the **Memory** window. The breakpoint is not highlighted—you can see, edit, and remove it in the breakpoint dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the Memory window, page 96*.
Set Data Log Breakpoint	Sets a breakpoint on the start address of a memory selection directly in the **Memory** window. The breakpoint is not highlighted—you can see, edit, and remove it in the breakpoint dialog box. The breakpoints you set in this window will be triggered by both read and write accesses—to change this, use the **Breakpoints** window. For more information, see *Data Log breakpoints, page 92* and *Getting started using data logging, page 198*.

Memory Save dialog box

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



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

Requirements

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

Zone

Selects a memory zone, see *C-SPY memory zones, page 121*.

Start address

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

End address

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

File format

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

Filename

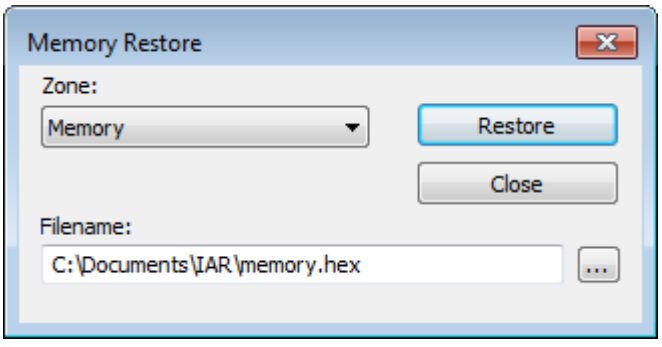
Specify the destination file to be used. A browse button is available.

Save

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

Memory Restore dialog box

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



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

Requirements

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

Zone

Selects a memory zone, see *C-SPY memory zones, page 121*.

Filename

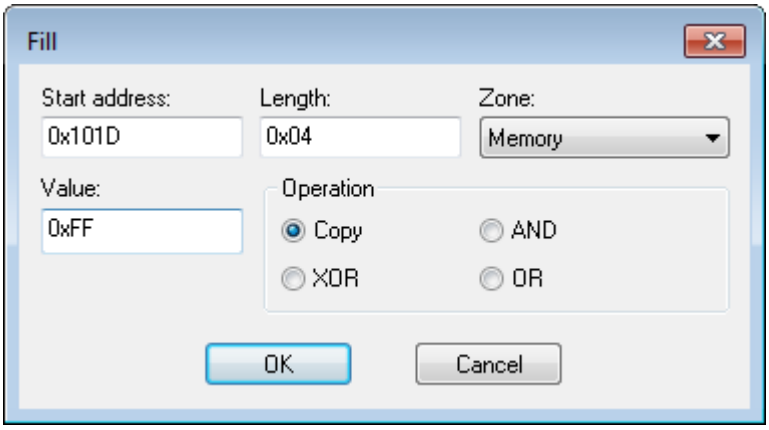
Specify the file to be read. A browse button is available.

Restore

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

Fill dialog box

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



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

Requirements

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

Start address

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

Length

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

Zone

Selects a memory zone, see *C-SPY memory zones, page 121*.

Value

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

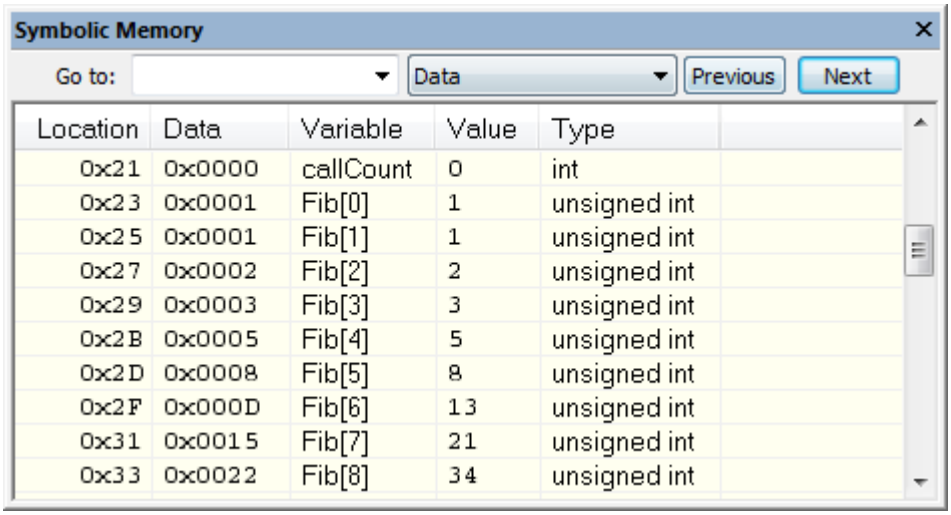
Operation

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.



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



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

See also *Editing in C-SPY windows, page 37*.

Requirements

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

Toolbar

The toolbar contains:

Go to The memory location or symbol you want to view.

Zone Selects a memory zone, see *C-SPY memory zones, page 121*.

Previous
Highlights the previous symbol in the display area.

Next
Highlights the next symbol in the display area.

Display area

This area contains these columns:

Location The memory address.

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

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

Value The value of the variable. This column is editable.

Type The type of the variable.

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

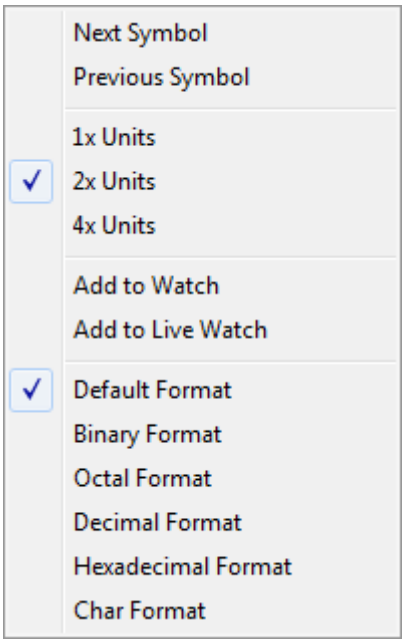
- Text that is dropped in the window is interpreted as symbols
- The scroll bar at the right-side of the window
- The **Next** and **Previous** toolbar buttons
- The **Go to** toolbar list box can be used for locating specific locations or symbols.



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

Context menu

This context menu is available:

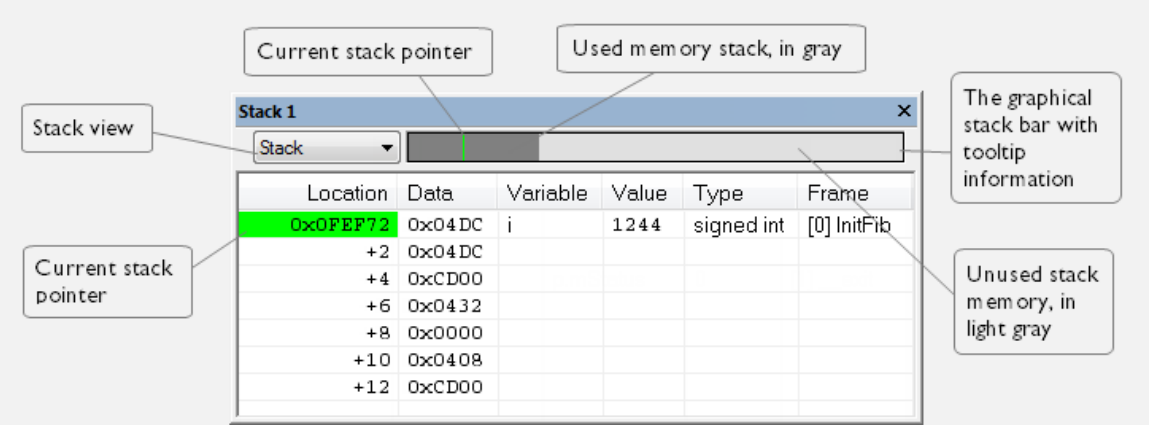


These commands are available:

Next Symbol	Highlights the next symbol in the display area.
Previous Symbol	Highlights the previous symbol in the display area.
1x Units	Displays the memory contents as single bytes. This applies only to rows that do not contain a variable.
2x Units	Displays the memory contents as 2-byte groups.
4x Units	Displays the memory contents as 4-byte groups.
Add to Watch	Adds the selected symbol to the **Watch** window.
Add to Live Watch	Adds the selected symbol to the **Live Watch** window.
Default format	Displays the memory contents in the default format.
Binary format	Displays the memory contents in binary format.
Octal format	Displays the memory contents in octal format.
Decimal format	Displays the memory contents in decimal format.
Hexadecimal format	Displays the memory contents in hexadecimal format.
Char format	Displays the memory contents in char format.

Stack window

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



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

This window retrieves information about the stack size and placement from the definition in the linker configuration file of the sections holding the stacks. The sections are described in the *IAR C/C++ Development Guide for Arm*. For applications that set up the stacks using other mechanisms, you can to override the default mechanism. Use one of the C-SPY command line option variants, see *--proc_stack_stack, page 450*.



By default, this window uses one physical breakpoint. For more information, see *Breakpoint consumers, page 93*.

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

Requirements

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

Toolbar

The toolbar contains:

Stack Selects which stack to view. This applies to cores with multiple stacks.

The graphical stack bar

Displays the state of the stack graphically.

The left end of the stack bar represents the bottom of the stack, in other words, the position of the stack pointer when the stack is empty. The right end represents the end of the memory address range reserved for the stack. The graphical stack bar turns red when the stack usage exceeds a threshold that you can specify.

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



Place the mouse pointer over the stack bar to get tooltip information about stack usage.

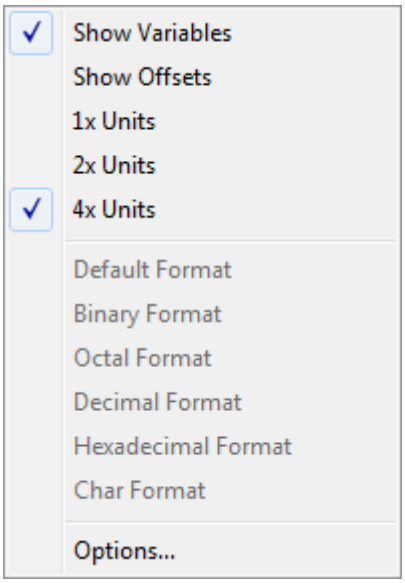
Display area

This area contains these columns:

Location	Displays the location in memory. The addresses are displayed in increasing order. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.
Data	Displays the contents of the memory unit at the given location. From the **Stack** window context menu, you can select how the data should be displayed—as a 1-, 2-, or 4-byte group of data.
Variable	Displays the name of a variable, if there is a local variable at the given location. Variables are only displayed if they are declared locally in a function, and located on the stack and not in registers.
Value	Displays the value of the variable.
Type	Displays the data type of the variable.
Frame	Displays the name of the function that the call frame corresponds to.

Context menu

This context menu is available:



These commands are available:

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

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

1x Units Displays the memory contents as single bytes.

2x Units Displays the memory contents as 2-byte groups.

4x Units Displays the memory contents as 4-byte groups.

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

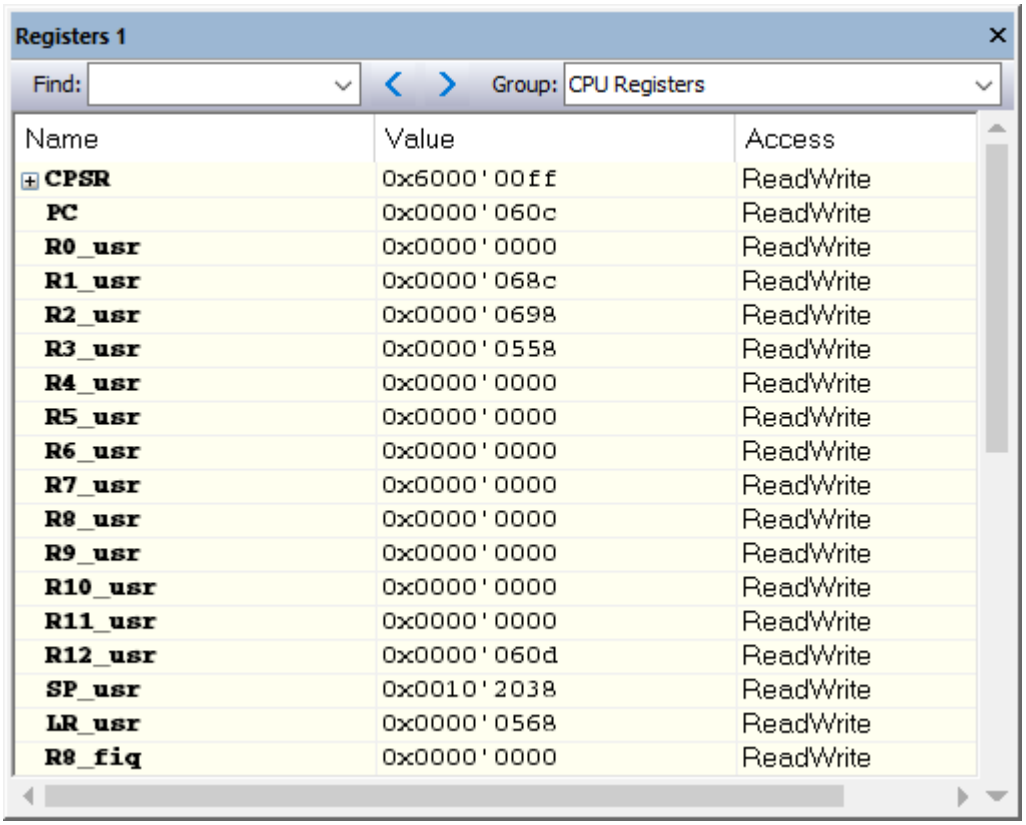
The display format setting affects different types of expressions in these ways:

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

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

Registers window

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



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

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

See also *Editing in C-SPY windows, page 37*.

To enable predefined register groups:

1. Select a device description file that suits your device, see *Selecting a device description file, page 35*. These files contain predefined register groups.
2. Display the registers of a register group by selecting it from the **Group** drop-down menu on the toolbar, or by right-clicking in the window and choosing **View Group** from the context menu.

   For information about creating your own user-defined register groups, see *Defining application-specific register groups, page 123*.

Requirements

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

Toolbar

The toolbar contains:

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

Group Selects which predefined register group to display. 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 142*.

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 **Registers** windows. The device description file contains a section that defines the special function registers and their groups.

Display area

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

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

Name The name of the register.

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

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

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

For the C-SPY Simulator (and some C-SPY hardware debugger drivers), these additional support registers are available in the CPU Registers group:

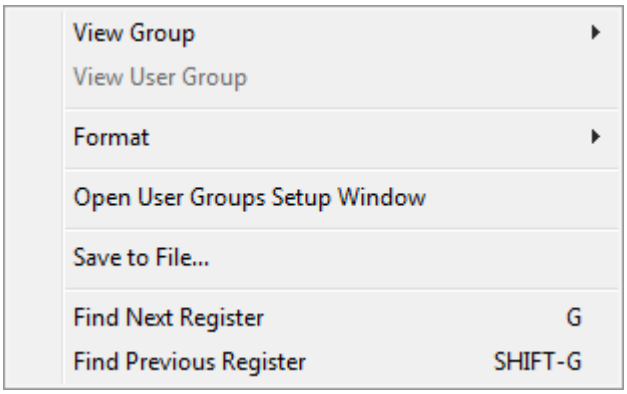
CYCLECOUNTER Cleared when an application is started or reset, and is incremented with the number of used cycles during execution.

CCSTEP Shows the number of used cycles during the last performed C/C++ source or assembler step.

CCTIMER1 and **CCTIMER2** Two *trip counts* that can be cleared manually at any given time. They are incremented with the number of used cycles during execution.

Context menu

This context menu is available:

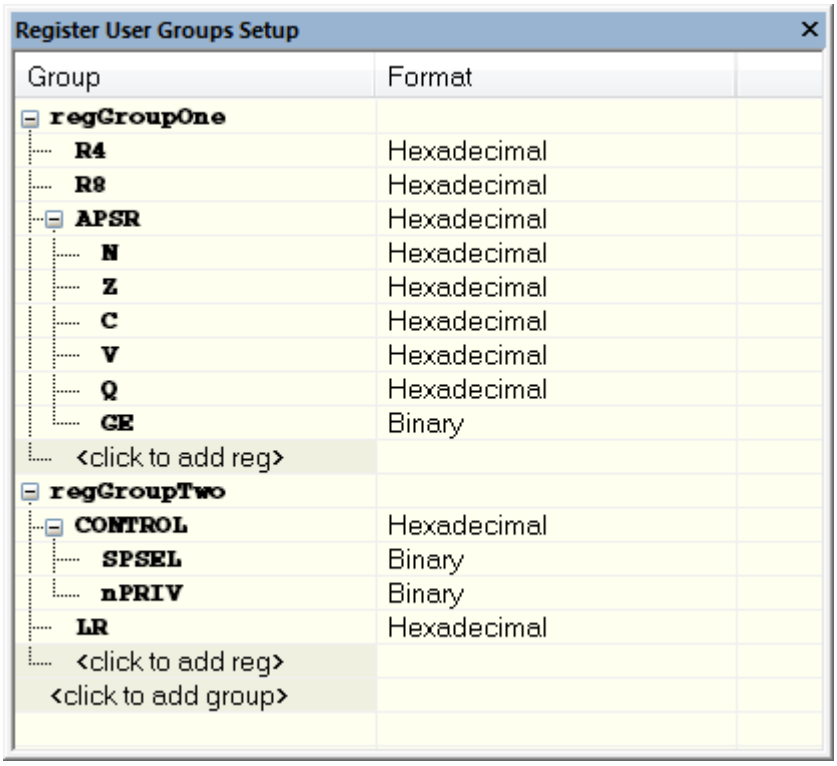


These commands are available:

View Group	Selects which predefined register group to display.
View User Group	Selects which user-defined register group to display. For information about creating your own user-defined register groups, see *Defining application-specific register groups, page 123*.
Format	Changes the display format for the contents of the register you clicked on. The display format setting affects different types of registers in different ways. Your selection of display format is saved between debug sessions.
Open User Groups Setup Window	Opens a window where you can create your own user-defined register groups, see *Register User Groups Setup window, page 141*.
Save to File	Opens a standard **Save** dialog box to save the contents of the window to a tab-separated text file.
Find Next Register	Finds the predefined register or register group that comes immediately after what your search found. After the last register was found, this search wraps around and finds the first register again.
Find Previous Register	Finds the matching predefined register or register group that comes immediately before what your search found. After the first register was found, this search wraps around and finds the last register again.

Register User Groups Setup window

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



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

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

Requirements

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

Display area

This area contains these columns:

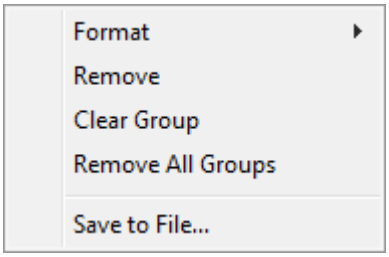
Group The names of register groups and the registers they contain. Clicking on `<click to add group>` or `<click to add reg>` and typing the name of a register group or register, adds new groups and registers, respectively. You can also drag a register name from another window in the IDE. Click a name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

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

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

Context menu

This context menu is available:

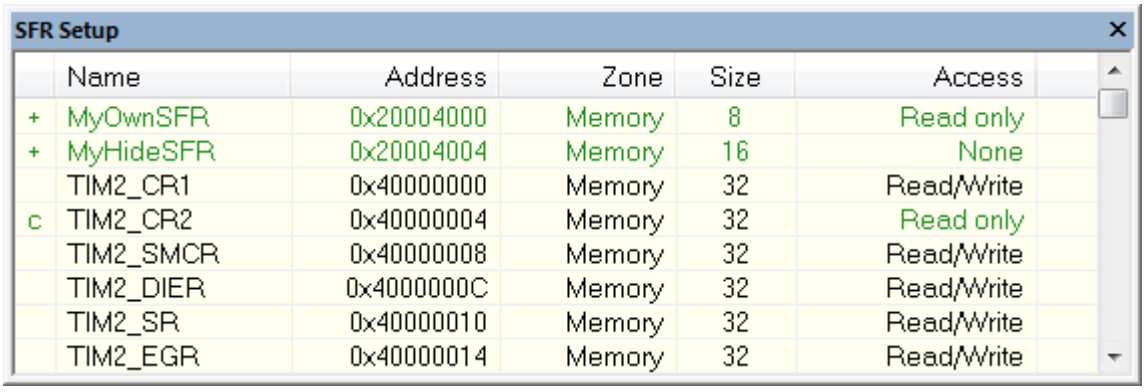


These commands are available:

Format	Changes the display format for the contents of the register you clicked on. The display format setting affects different types of registers in different ways. Your selection of display format is saved between debug sessions.
Remove	Removes the register or group you clicked on.
Clear Group	Removes all registers from the group you clicked on.
Remove All Groups	Deletes all user-defined register groups from your project.
Save to File	Opens a standard **Save** dialog box to save the contents of the window to a tab-separated text file.

SFR Setup window

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



This window displays ....

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

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

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

Requirements

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

Display area

This area contains these columns:

Status	A character that signals the status of the SFR, which can be one of: blank, a factory-defined SFR. **C**, a factory-defined SFR that has been modified. **+**, a custom-defined SFR. **?**, an SFR that is ignored for some reason. An SFR can be ignored when a factory-defined SFR has been modified, but the SFR is no longer available, or it is located somewhere else or with a different size. Typically, this might happen if you change to another device.
Name	A unique name of the SFR.
Address	The memory address of the SFR.
Zone	Selects a memory zone, see *C-SPY memory zones, page 121*.
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`.

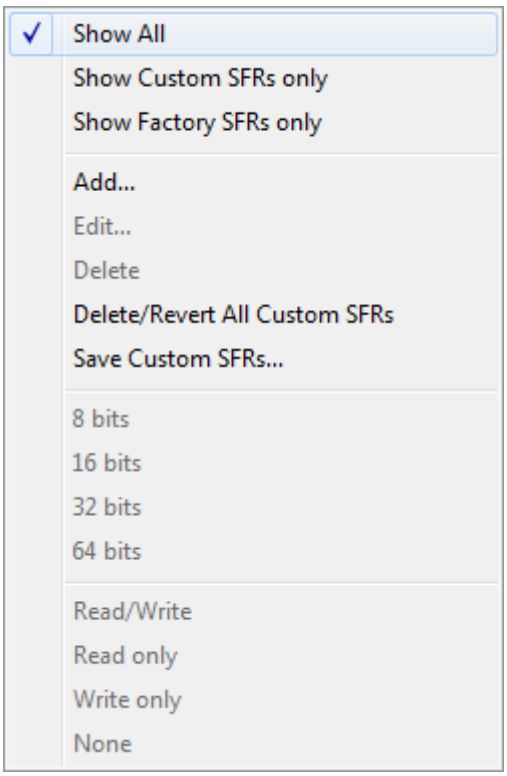
You can click a column header to sort the SFRs according to the column property.

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:

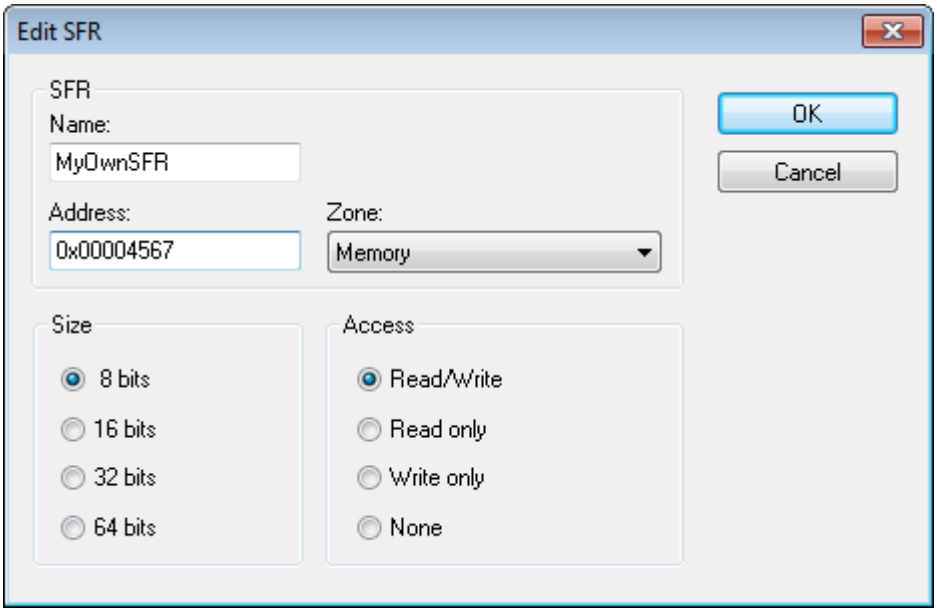


These commands are available:

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 145*.
Edit	Displays the **Edit SFR** dialog box where you can edit an SFR, see *Edit SFR dialog box, page 145*.
Delete	Deletes an SFR. This command only works on custom-defined SFRs.
Delete/Revert All Custom SFRs	Deletes all custom-defined SFRs and reverts all modified factory-defined SFRs to their factory settings.
Save Custom SFRs	Opens a standard **Save** dialog box to save all custom-defined SFRs.
8|16|32|64 bits	Selects display format for the selected SFR, which can be **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.
Read/Write|Read only|Write only|None	Selects the access type of the selected SFR, which can be **Read/Write**, **Read only**, **Write only**, or **None**. Note that for factory-defined SFRs, the default access type is indicated.

Edit SFR dialog box

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



Definitions of the SFRs are retrieved from the device description file in use. Use this dialog box to either modify these factory-defined definitions or define new SFRs. See also *SFR Setup window, page 142*.

Requirements

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

Name

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

Address

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

Zone

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

Size

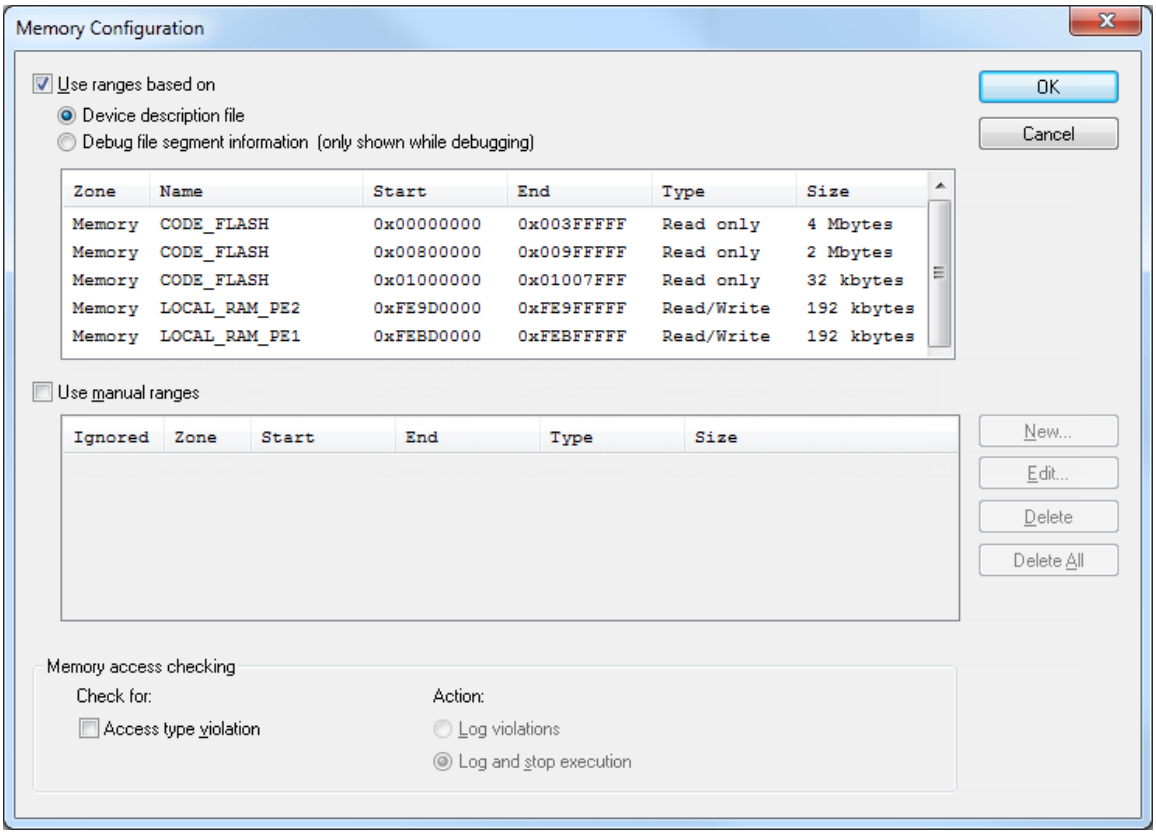
Selects the size of the SFR. Choose between **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.

Access

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

Memory Configuration dialog box for the C-SPY simulator

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



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

See also *Memory configuration for the C-SPY simulator, page 122*.

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 35*. 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 121*.
Name	The name of the memory address range.
Start	The start address for the memory address range, in hexadecimal notation.
End	The end address for the memory address range, in hexadecimal notation.
Type	The access type of the memory address range.
Size	The size of the memory address range.

Use manual ranges

Specify your own ranges manually via the **Edit Memory Range** dialog box. To open this dialog box, click **New** to specify a new memory address range, or select an existing memory address range and click **Edit** to modify it. For more information, see *Edit Memory Range dialog box for the C-SPY simulator, page 148*.

The ranges you define manually are saved between debug sessions.

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

Memory access checking

Check for determines what to check for:

- **Access type violation**.

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

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

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

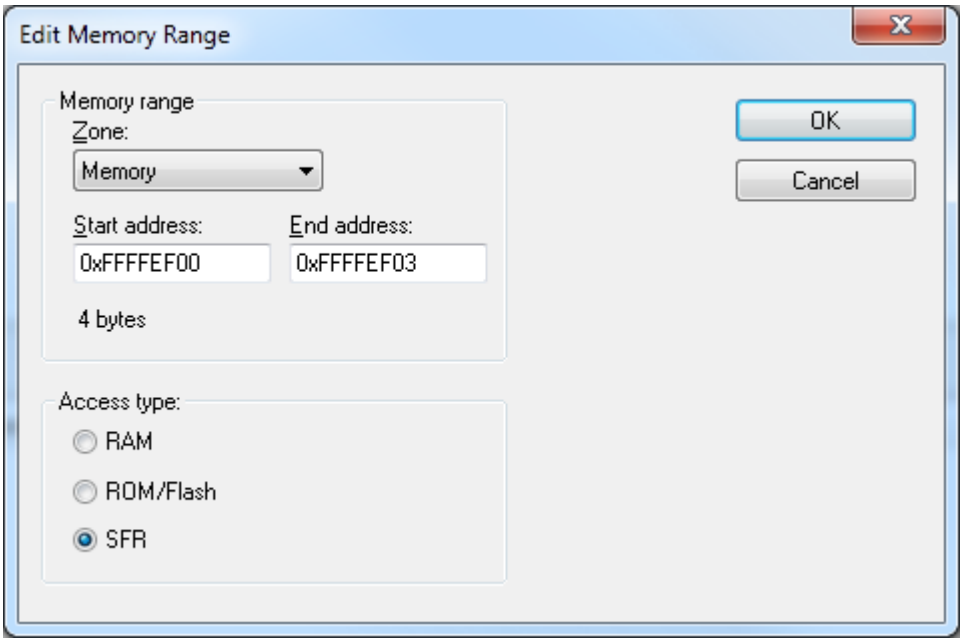
Buttons

These buttons are available for the manual ranges:

New	Opens the **Edit Memory Range** dialog box, where you can specify a new memory address range and associate an access type with it, see *Edit Memory Range dialog box for the C-SPY simulator, page 148*.
Edit	Opens the **Edit Memory Range** dialog box, where you can edit the selected memory address range. See *Edit Memory Range dialog box for the C-SPY simulator, page 148*.
Delete	Deletes the selected memory address range definition.
Delete All	Deletes all defined memory address range definitions.

Edit Memory Range dialog box for the C-SPY simulator

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



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

See also *Memory Configuration dialog box for the C-SPY simulator, page 146*

Requirements

The C-SPY simulator.

Memory range

Defines the memory address range specific to your device:

Zone	Selects a memory zone, see *C-SPY memory zones, page 121*.
Start address	Specify the start address for the memory address range, in hexadecimal notation.
End address	Specify the end address for the memory address range, in hexadecimal notation.

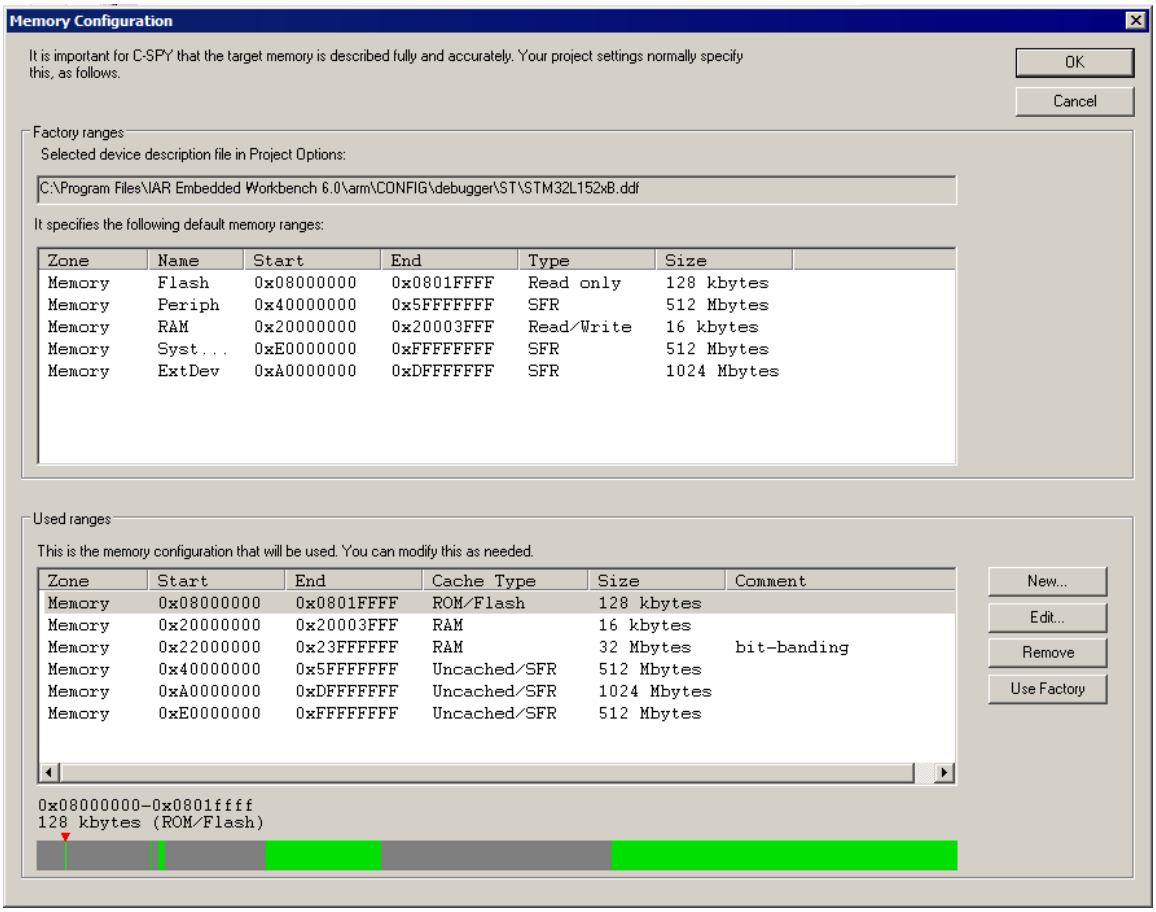
Access type

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

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

Memory Configuration dialog box for C-SPY hardware debugger drivers

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



C-SPY uses a default memory configuration based on information retrieved from the device description file 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 35*.

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.

See also *Memory configuration for C-SPY hardware debugger drivers, page 122*.

Requirements

One of these alternatives:

- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver
- The C-SPY CMSIS-DAP driver
- The C-SPY GDB Server driver
- The C-SPY TI MSP-FET driver
- The C-SPY TI Stellaris driver
- The C-SPY TI XDS driver

Factory ranges

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

Zone The memory zone, see *C-SPY memory zones, page 121*.
Name The name of the memory address range.
Start The start address for the memory address range, in hexadecimal notation.
End The end address for the memory address range, in hexadecimal notation.
Type The access type of the memory address range.
Size The size of the memory address range.

Used ranges

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

Zone Selects a memory zone, see *C-SPY memory zones, page 121*.
Start The start address for the memory address range, in hexadecimal notation.

End	The end address for the memory address range, in hexadecimal notation.
Cache Type	The cache type of the memory address range.
Size	The size of the memory address range.
Comment	Memory area information.

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

Graphical bar

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

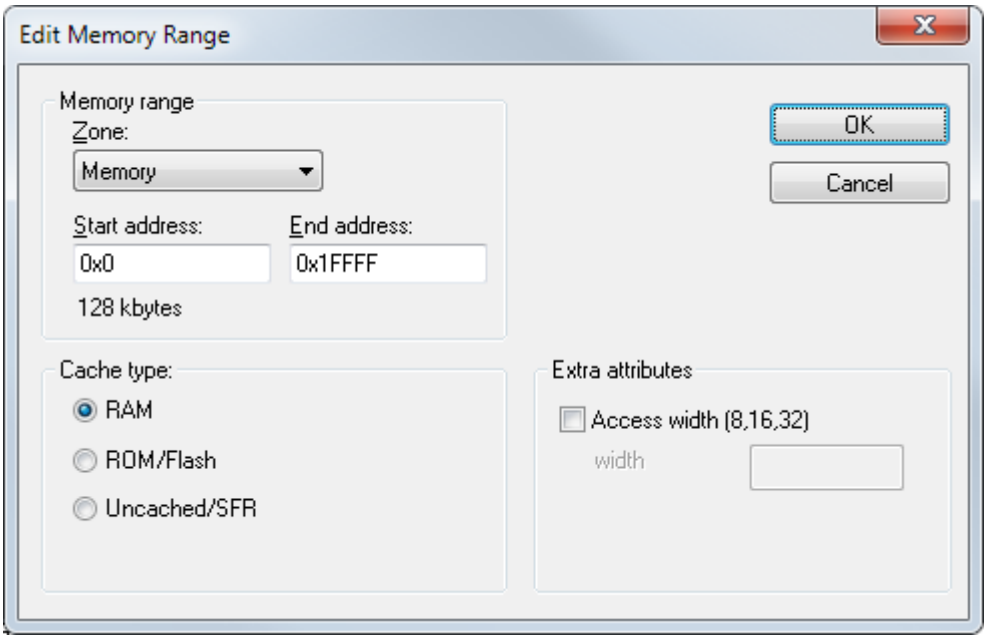
Buttons

These buttons are available for manual ranges:

New	Opens the **Edit Memory Range** dialog box, where you can specify a new memory address range and associate a cache type with it, see *Edit Memory Range dialog box for C-SPY hardware debugger drivers, page 151*.
Edit	Opens the **Edit Memory Range** dialog box, where you can edit the selected memory address area. See *Edit Memory Range dialog box for C-SPY hardware debugger drivers, page 151*.
Remove	Removes the selected memory address range definition.
Use Factory	Restores the list of used ranges to the factory ranges.

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

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



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

See also *Memory configuration for C-SPY hardware debugger drivers, page 122*.

Requirements

One of these alternatives:

- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver
- The C-SPY CMSIS-DAP driver
- The C-SPY GDB Server driver
- The C-SPY TI MSP-FET driver
- The C-SPY TI Stellaris driver
- The C-SPY TI XDS driver

Memory range

Defines the memory address range specific to your device:

Zone	Selects a memory zone, see *C-SPY memory zones, page 121*.
Start address	Specify the start address for the memory address range, in hexadecimal notation.
End address	Specify the end address for the memory address range, in hexadecimal notation.

Cache type

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

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

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

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

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

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

If required, you can disable caching—choose *Driver*>**Disable Debugger Cache**.

Extra attributes

Provides extra attributes.

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

This option might not be available in the C-SPY driver you are using.

Part 2. Analyzing your application

Part 2. Analyzing your application

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

- *Trace, page 155*
- *The application timeline, page 195*
- *Profiling, page 224*
- *Code coverage, page 233*
- *Performance monitoring, page 238*
- *Power debugging, page 243*
- *C-RUN runtime error checking, page 263*

Trace

Contents

Introduction to using trace 155
Reasons for using trace 155
Briefly about trace 156
Requirements for using trace 157
Collecting and using trace data 158
Getting started with ETM trace 158
Getting started with SWO trace 159
Getting started with MTB trace 159
Setting up concurrent use of ETM and SWO 160
Trace data collection using breakpoints 160
Searching in trace data 160
Browsing through trace data 161
Reference information on trace 161
ETM Trace Settings dialog box (I-jet) 161
ETM Trace Settings dialog box (J-Link/J-Trace) 163
SWO Trace Window Settings dialog box 165
SWO Configuration dialog box 167
Trace window 170
Function Trace window 178
Trace Start Trigger breakpoint dialog box 179
Trace Stop Trigger breakpoint dialog box 180
Trace Start breakpoints dialog box (I-jet and CMSIS-DAP) 181
Trace Stop breakpoints dialog box (I-jet and CMSIS-DAP) 183
Trace Filter breakpoints dialog box (I-jet) 184
Trace Start breakpoints dialog box (J-Link/J-Trace) 185
Trace Stop breakpoints dialog box (J-Link/J-Trace) 187
Trace Filter breakpoints dialog box (J-Link/J-Trace) 189
Find in Trace dialog box 191
Find in Trace window 193
Trace Save dialog box 193

INTRODUCTION TO USING TRACE

See also:

- *Getting started using data logging, page 198*
- *Getting started using event logging, page 199*
- *Power debugging, page 243*
- *Getting started using interrupt logging, page 311*
- *Profiling, page 224*

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 filters

By using trace trigger and 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 Trigger and Trace Stop Trigger 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.

Briefly about trace

To use trace in C-SPY requires that your target system can generate trace data. Once generated, C-SPY can collect it and you can visualize and analyze the data in various windows and dialog boxes.

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

- 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
- For some cores and devices, 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 or ETR). 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. For ETB trace, the trace buffer has a designated memory area with a predefined size.

ETR trace

ETR (Embedded Trace Router) trace is a trace variant that uses an on-chip trace buffer. For ETR trace, the trace buffer shares the RAM memory with your application code and is configured in the project options.

MTB trace

MTB trace (Micro Trace Buffer) is a simplified trace variant that 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 some devices based on the Cortex-M0+ and M23 cores.

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 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**. 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 driver, the C-SPY J-Link/J-Trace driver, the C-SPY ST-LINK driver, or the C-SPY TI XDS 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 either of these buttons are green, it means that the corresponding trace hardware is generating trace data. For detailed tooltip information about which C-SPY features have requested trace data generation, simply point at the button with the mouse pointer. 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 for some cores and devices. If supported, there are no further specific requirements.



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 ETMv3 trace, or earlier, you need one of these combinations:

- An I-jet Trace in-circuit debugging probe and a device that supports ETM via a physical trace port. Make sure to use the C-SPY I-jet driver.
- An I-jet or I-jet Trace in-circuit debugging probe and a device that supports ETM via ETB or ETR. The debug probe reads ETM data from the ETB or ETR buffer. Make sure to use the C-SPY I-jet driver.
- A J-Link or J-Trace debug probe and a device that supports ETM via ETB or ETR. The debug probe reads ETM data from the ETB or ETR buffer. Make sure to use the C-SPY J-Link/J-Trace driver.

- A J-Trace debug probe and a device that supports ETM via a physical trace port. Make sure to use the C-SPY J-Link/J-Trace driver.

To use ETMv4 trace, you need one of these combinations:

- An I-jet Trace in-circuit debugging probe and a device that supports ETM via a physical trace port. Make sure to use the C-SPY I-jet driver.
- An I-jet or I-jet Trace in-circuit debugging probe and a device that supports ETM via ETB or ETR. The debug probe reads ETM data from the ETB or ETR buffer. Make sure to use the C-SPY I-jet driver.

For more information, see the *IAR Debug probes User Guide for I-jet and I-jet Trace* and the J-Link/J-Trace documentation available at *www.segger.com*, respectively.



For ETB and ETR, the device support must implement specific trace support for either ETB or ETR. Specific setup sequences might also be needed in the corresponding device support macro files (`.dmac`).

Requirements for using MTB (Micro Trace Buffer) trace

To use MTB trace, you need an M0+ or M23 device with MTB, and one of these alternatives:

- An I-jet in-circuit debugging probe
- The C-SPY CMSIS-DAP driver and a device that supports CMSIS-DAP
- A J-Link debug probe

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, an ST-LINK, or a TI XDS 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 for ETM, ETB, ETR, and PTM trace, but not for SWO or MTB trace.

Different processors, cores, and debug probes might limit the number of available trace triggers and filters.

COLLECTING AND USING TRACE DATA

Getting started with ETM trace

1. Before you start C-SPY, you must set up the trace port for your device. 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.

   Note that the pins used on the hardware for the trace signals cannot be used by your application.
2. If your device uses ETR, specify the ETR trace buffer memory layout on the **Project>Options>Debugger>Extra options** tab. The available parameters are:
   - `--macro_param etrram`

     The start of the RAM area for trace storage, for example, `0x040A0000`. If not specified, ETB trace will be used instead of ETR trace.
   - `--macro_param etrsize`

     The size of the RAM area for trace storage, for example, `0x60000`.

- `--macro_param etrport`

  The access port used to access the ETR RAM, for example, `1`. If not specified and if ETR is enabled, the port that is used to access the core will be used.

3. 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.
4.  Open the **Trace** window—available from the driver-specific menu—and click the **Activate** button to enable collecting trace data.
5. 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.
6. 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, see *Trace window, page 170*.

Getting started with SWO trace

1. Before you start C-SPY, choose **Project>Option>*Driver***

   Click the **Setup** tab or the **Connection** tab, respectively, and choose **Interface>SWD**.
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 165*.
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 167*.

   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 the window, see *Trace window, page 170*.

Getting started with MTB trace

1. Before you start C-SPY, use *one* of these alternatives to configure the MTB trace buffer:
   - Define an array in RAM called `__mtb_trace_buffer` in your application, which the debugger will use as the trace buffer.
   - Define the two symbols `MTB_TRACE_BUFFER$$Base` and `MTB_TRACE_BUFFER$$Limit` in the linker configuration file (`.icf`). The debugger will use this information as the address and the size of the trace buffer.
   - Let the debugger automatically find some unused RAM space and use that as the trace buffer.
2. Start C-SPY.
3. Open the **Trace** window—available from the driver-specific menu—and click the **Activate** button to enable collecting trace data.
4. Start the execution. When the execution stops, for instance because a breakpoint is triggered, trace data is displayed in the **Trace** window. For more information, see *Trace window, page 170*.

Setting up concurrent use of ETM and SWO

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

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

Trace data collection using breakpoints

A convenient way to collect trace data between two execution points is to start and stop the data collection using dedicated breakpoints.

Choose between these alternatives:

- In the editor or **Disassembly** window, position your insertion point, right-click, and toggle a **Trace Start Trigger** or **Trace Stop Trigger** breakpoint from the context menu.
- In the **Breakpoints** window, choose **New Breakpoint>Trace Start Trigger**, **Trace Stop Trigger**, or **Trace Filter** from the context menu.
- The C-SPY system macros `__setTraceStartBreak` and `__setTraceStopBreak` can also be used.

For more information about these breakpoints, see *Trace Start Trigger breakpoint dialog box, page 179* and *Trace Stop Trigger breakpoint dialog box, page 180*, respectively.

Using the trace triggers and trace filters:

1. Use the **Trace Start Trigger** dialog box to set a start condition—a start trigger—to start collecting trace data.
2. Use the **Trace Stop Trigger** 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 also in browse mode in the **Disassembly** window, where the trace marks for your trace triggers and trace filters are also visible.
8. If you have set a trace filter, the trace data collection is performed while the condition is true plus some further instructions. When viewing the trace data and looking for a certain data access, remember that the access took place one instruction earlier.

Searching in trace data

When you have collected trace data, you can perform searches in the collected data to locate the parts of your code or data that you are interested in, for example, a specific interrupt or accesses of a specific variable.

You specify the search criteria in the **Find in Trace** dialog box and view the result in the **Find in Trace** window.



The **Find in Trace** 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 191*.
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 193*.

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.

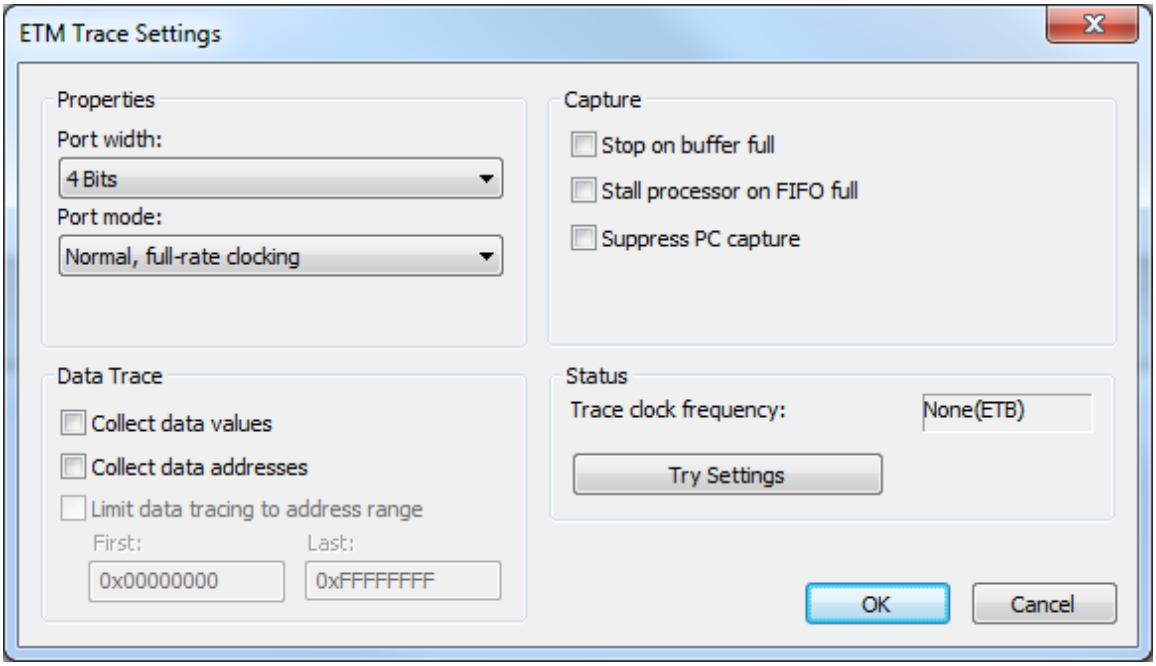
The selected item turns yellow and the source and **Disassembly** windows will highlight the corresponding location. You can now move around in the trace data using the up and down arrow keys, or by scrolling and clicking—the source and **Disassembly** windows will be updated to show the corresponding location. This is like stepping backward and forward through the execution history.

Double-click again to leave browse mode.

REFERENCE INFORMATION ON TRACE

ETM Trace Settings dialog box (I-jet)

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 157*
- *Getting started with ETM trace, page 158*

Requirements

One of these alternatives:

- The C-SPY I-jet driver
- The C-SPY CMSIS-DAP 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

Data Trace

Selects what type of trace data you want C-SPY to collect. Data trace is only available for Arm7/Arm9/Arm11-based devices with ETM data trace. 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.

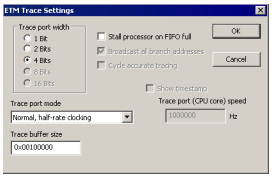
Status

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 157*
- *Getting started with ETM trace, page 158*

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



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 `0xFFFFF`, and for Cortex-M3 the maximum number is `0x3FFFFF`.

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.



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.



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.



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

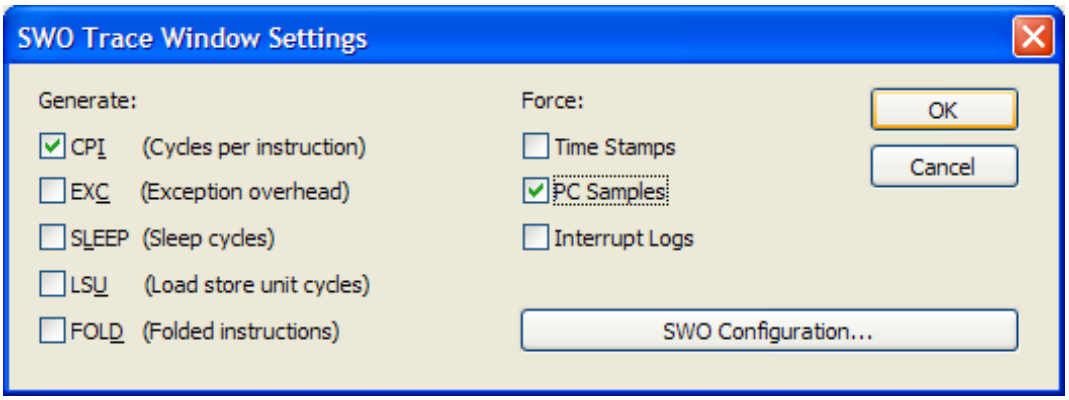
Makes the **Trace** window display seconds instead of cycles in the **Index** column. To make this possible, you must also specify the appropriate speed for your CPU in the **Trace port (CPU core) speed** text box.



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 C-SPY driver menu, 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 167*.

Requirements

One of these alternatives:

- The C-SPY I-jet driver and an I-jet or I-jet Trace in-circuit debugging probe
- The C-SPY J-Link/J-Trace driver and a J-Link/J-Trace JTAG/SWD probe
- The C-SPY ST-LINK driver and an ST-LINK JTAG/SWD probe
- The C-SPY TI XDS driver and a TI XDS 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 167*, 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 306*.

ITM Log Forces the generation of ITM logs to the **SWO Trace** window.

This option only applies to I-jet.

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.

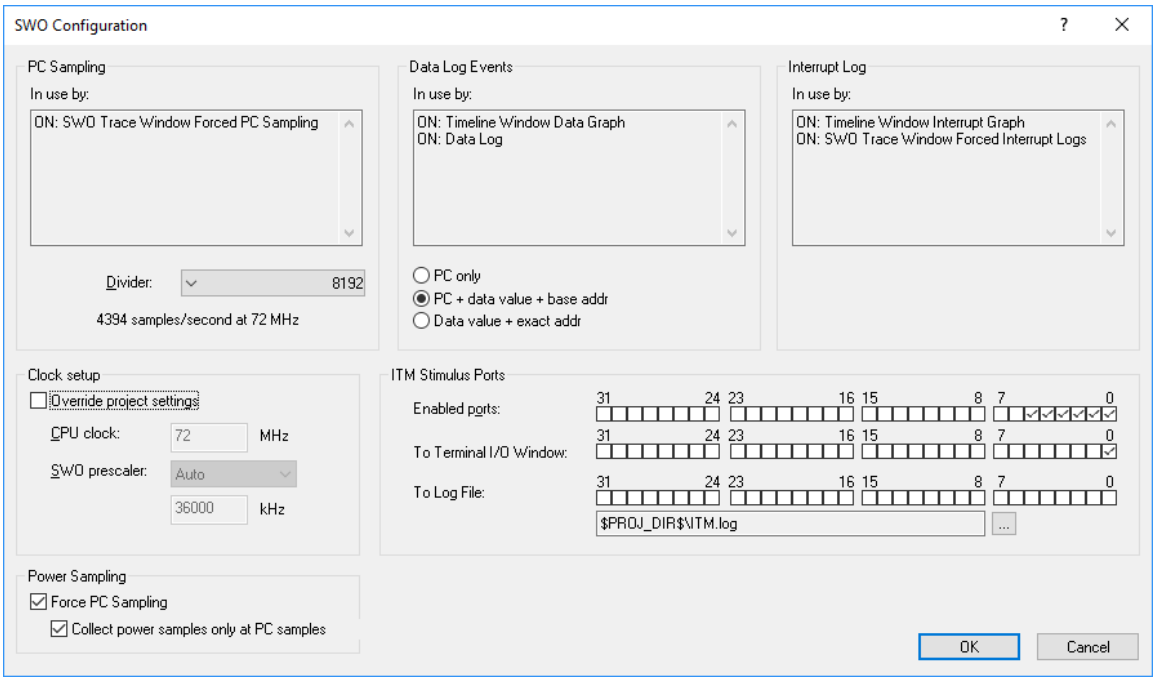
SWO Configuration

Displays the **SWO Configuration** dialog box where you can configure the hardware's generation of trace data. See *SWO Configuration dialog box, page 167*.

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.



This figure reflects the C-SPY I-jet driver.

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 159*.

Requirements

One of these alternatives:

- The C-SPY I-jet driver and an I-jet or I-jet Trace in-circuit debugging probe
- The C-SPY J-Link/J-Trace driver and a J-Link/J-Trace JTAG/SWD probe
- The C-SPY ST-LINK driver and an ST-LINK JTAG/SWD probe
- The C-SPY TI XDS driver and a TI XDS 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 only applies to I-jet.

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 306*.

Override project default

Overrides the **CPU clock** and the **SWO clock** default values on the **Project>Options>J-Link/J-Trace>Setup** page, on the **Project>Options>ST-Link>Communication** page, or on the **Project>Options>TI XDS>Communication** page, 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, ST-LINK, and TI XDS, 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 only applies to I-jet.

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.

Power Sampling

Force PC Sampling	Forces PC sampling, even when no other C-SPY features have requested it. This makes it possible to associate PC values with power samples.
Collect power samples only at PC samples	Collects power samples only at PC samples, so that every power sample can be associated with a program counter position.

Trace window

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

This window displays the collected trace data.



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

The content of the **Trace**window depends on the C-SPY driver you are using and the trace support of your debug probe.

See also *Collecting and using trace data, page 158*.

Requirements

One of these alternatives:

- The C-SPY Simulator. (Not available for all cores and devices.)
- A CMSIS-DAP probe
- 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
- A TI XDS probe

Trace toolbar

The toolbar in the **Trace** window contains:

Enable/Disable



Enables and disables collecting and viewing trace data in this window.

Clear trace data



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

Toggle source



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

Browse



Toggles browse mode on or off for a selected item in the **Trace** window, see *Browsing through trace data, page 161*.

Find



Displays a dialog box where you can perform a search, see *Find in Trace dialog box, page 191*.

Save



In the **ETM Trace** and **SWO Trace** windows, this button displays the **Trace Save** dialog box, see *Trace Save dialog box, page 193*.

In the C-SPY I-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 163* and *ETM Trace Settings dialog box (I-jet), page 161*.

In the **SWO Trace** window this button displays the **SWO Trace Window Settings** dialog box, see *SWO Trace Window Settings dialog box, page 165*.

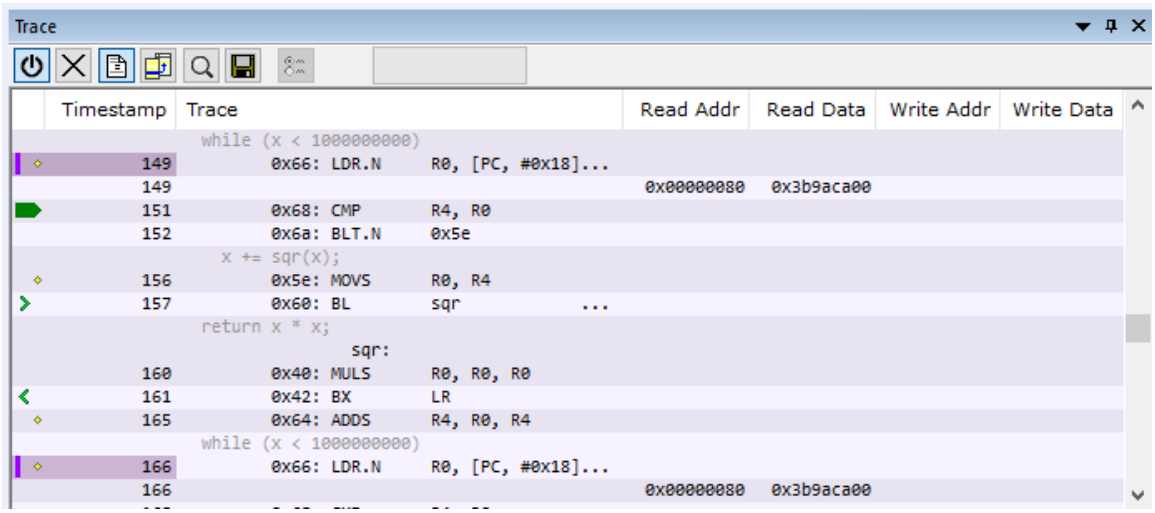
Progress bar



When a large amount of trace data has been collected, there might be a delay before all of it has been processed and can be displayed. The progress bar reflects that processing.

Display area (in the C-SPY simulator)

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

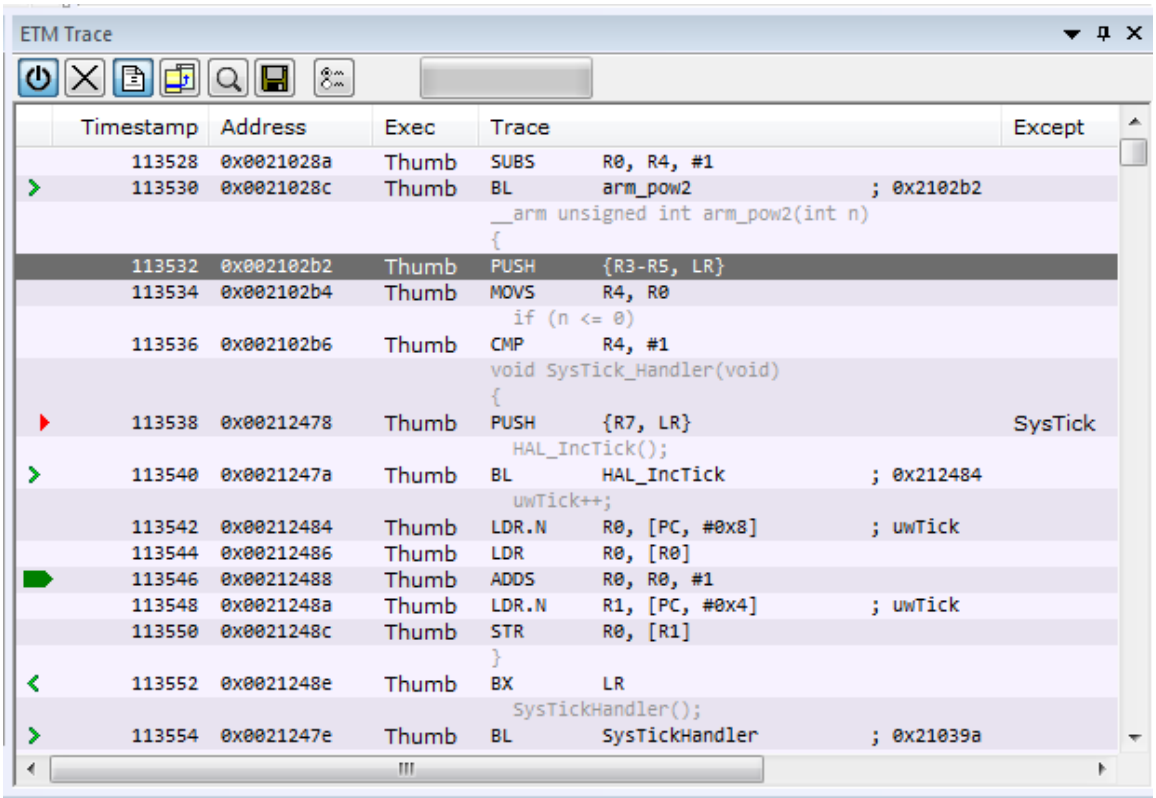


This area contains these columns for the C-SPY simulator:

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 in the C-SPY hardware debugger drivers)

This area displays a collected sequence of executed machine instructions and other trace data.



Data trace is only available for the C-SPY I-jet driver or CMSIS-DAP driver, when using Arm7/Arm9/Arm11-based devices with ETM data trace.

This area contains these columns. Note that some columns depend on the C-SPY driver, the CPU, and the probe you are using.

The leftmost column contains identifying icons to simplify navigation within the buffer:

	The yellow diamond indicates the trace execution point, marking when target execution has started.
	The right green arrow indicates a call instruction.
	The left green arrow indicates a return instruction.
	The dark green bookmark indicates a navigation bookmark.
	The red arrow indicates an interrupt.
	The violet bar indicates the results of a search.

Timestamp The internal I-jet Trace timestamp.

Address The address of the instruction associated with the trace frame.

Opcode The operation code of the instruction associated with the trace frame. After the hexadecimal value, extra information can be displayed:

`x2` — if two instructions were executed

`C` — if the instruction was read from the I-Cache

Exec	The execution mode—`Arm`, `Thumb`, or `NoExec`.
Trace	The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.
Except	The type of exception, when it occurs.
Access	The access type of the instruction associated with the trace frame. `DMA` stands for DMA transfer. The address and data information shows which transfer that was performed.
Data address	The data trace address.
Data value	The data trace value.
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:

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 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, including the values of the selected Trace Events counters, or the number of the comparator (hardware breakpoint) used for the Data Log breakpoint.

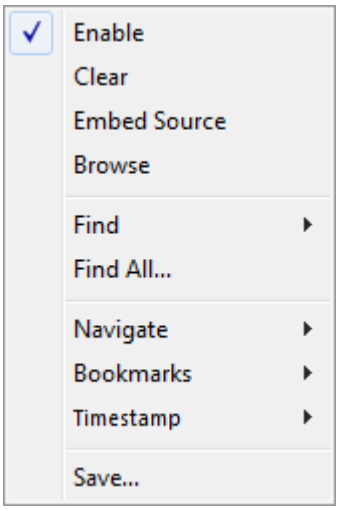
A red-colored row indicates that the previous row and the red row are not consecutive. This means that there is a gap in the collected trace data, for example because trace data has been lost due to an overflow.



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.

Context menu

This context menu is available:



The contents of this menu are dynamic and depend on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands. Note that the shortcuts to the submenu commands do not use the Ctrl key.

These commands are available:

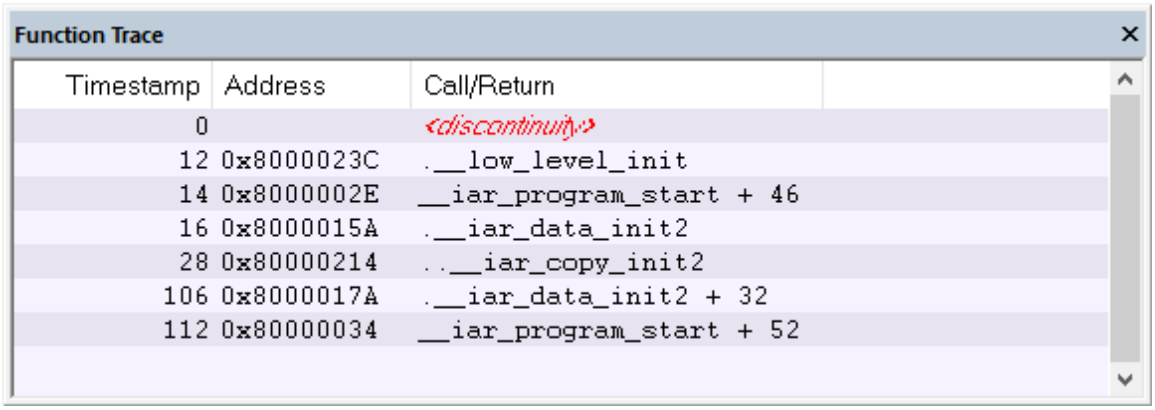
Enable	Enables and disables collecting and viewing trace data in this window.
Clear	Clears the trace buffer. Both the **Trace** window and the **Function Trace** window are cleared.
Embed source	Toggles the **Trace** column between showing only disassembly or disassembly together with the corresponding source code.
Browse	Toggles browse mode on or off for a selected item in the **Trace** window, see *Browsing through trace data, page 161*.
Find>Find (F)	Displays a dialog box where you can perform a search in the **Trace** window, see *Find in Trace dialog box, page 191*. The contents of the window will scroll to display the first match.
Find>Find Next (G)	Finds the next occurrence of the specified string.
Find>Find Previous (Shift+G)	Finds the previous occurrence of the specified string.
Find>Clear (Shift+F)	Removes all search highlighting in the window.
Find All	Displays a dialog box where you can perform a search in the **Trace** window, see *Find in Trace dialog box, page 191*. The search results are displayed in the **Find in Trace** window—available by choosing the **View>Messages** command, see *Find in Trace window, page 193*.
Navigate>After Current Loop (L)	Identifies the selected program counter and scans the trace data forward, collecting program counters, until it finds the same address again. It has now detected a loop. (Loops longer than 1000 instructions are not detected.) Then it navigates forward until it finds a program counter that is not part of the collected set. This is useful for navigating out of many iterations of an idle or polling loop.

Navigate>Before Current Loop (Shift+L)	Behaves as **After Current Loop**, but navigates backward out of the loop.
Navigate>After Current Function (U)	Navigates to the next unmatched return instruction. This is similar to stepping out of the current function.
Navigate>Before Current Function (Shift+U)	Navigates to the closest previous unmatched call instruction.
Navigate>Next Statement (S)	Navigates to the next instruction that belongs to a different C statement than the starting point. It skips function calls, i.e. it tries to reach the next statement in the starting frame.
Navigate>Previous Statement (Shift+S)	Behaves as **Next statement**, but navigates backward to the closest previous different C statement.
Navigate>Next on Same Address (A)	Navigates to the next instance of the starting program counter address, typically to the next iteration of a loop.
Navigate>Previous on Same Address (Shift+A)	Navigates to the closest previous instance of the starting program counter address.
Navigate>Next Interrupt (I)	Navigates to the next interrupt entry. (To then find the matching interrupt exit, follow up with **After Current Function**.)
Navigate>Previous Interrupt (Shift+I)	Navigates to the closest previous interrupt entry.
Navigate>Next Execution Start Point (E)	Navigates to the next point where the CPU was started, for example places where the application stopped at breakpoints, or was stepped.
Navigate>Previous Execution Start Point (Shift+E)	Navigates to the closest previous point where the CPU was started.
Navigate>Next Discontinuity (D)	Navigates to the next discontinuity in the trace data.
Navigate>Previous Discontinuity (Shift+D)	Navigates to the closest previous discontinuity in the trace data.
Bookmarks>Toggle (+)	Adds a new navigation bookmark or removes an existing bookmark.
Bookmarks>Goto Next (B)	Navigates to the next navigation bookmark.
Bookmarks>Goto Previous (Shift+B)	Navigates to the closest previous navigation bookmark.
Bookmarks>Clear All	Removes all navigation bookmarks.
Bookmarks>*location* (0–9)	At the bottom of the submenu, the ten most recently defined bookmarks are listed, with a shortcut key each from 0–9.
Timestamp>Set as Zero Point (Z)	Sets the selected row as a reference “zero” point in the collected sequence of trace data. The count of rows in the **Trace** window will show this row as `0` and recalculate the timestamps of all other rows in relation to this timestamp.

Timestamp>Go to Zero Point (Shift+Z)	Navigates to the reference “zero” point in the collected sequence of trace data (if you have set one).
Timestamp>Clear Zero Point	Removes the reference “zero” point from the trace data and restores the original timestamps of all rows.
Save	In the **ETM Trace** and **SWO Trace** windows, this button displays the **Trace Save** dialog box, see *Trace Save dialog box, page 193*. In the C-SPY I-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.

Function Trace window

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



This window displays a subset of the trace data displayed in the **Trace** window. Instead of displaying all rows, the **Function Trace** window shows:

- The functions called or returned to, instead of the traced instruction
- The corresponding trace data.

Requirements

One of these alternatives:

- The C-SPY Simulator. (Not available for all cores and devices.)
- 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
- A TI XDS probe

Display area

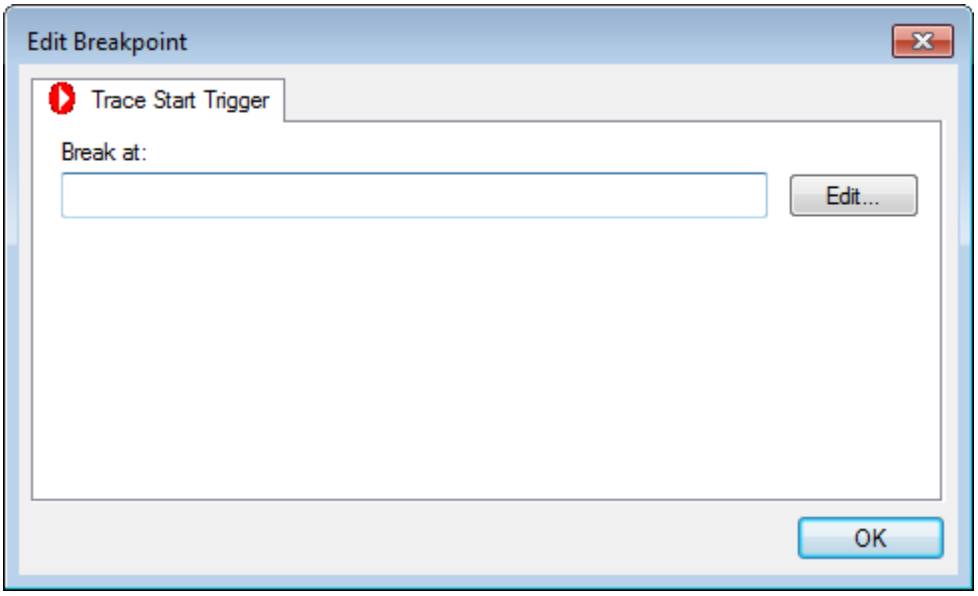
There are two sets of columns available, and which set is used in your debugging system depends on the debug probe and which trace sources that are available:

- For some combinations of debug probe and trace sources, the available columns are the same as in the **Trace** window, see *Trace window, page 170*.
- For the simulator, I-jet Trace, and I-jet and depending on the trace source, these columns are available:

Timestamp	The number of cycles elapsed to this point according to the timestamp in the debug probe.
Address	The address of the executed instruction.
Call/Return	The function that was called or returned to.

Trace Start Trigger breakpoint dialog box

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



Use this dialog box to set a Trace Start Trigger breakpoint where you want to start collecting trace data. If you want to collect trace data only for a specific range, you must also set a Trace Stop Trigger breakpoint where you want to stop collecting data.

See also *Trace Stop Trigger breakpoint dialog box, page 180* and *Trace data collection using breakpoints, page 160*.

To set a Trace Start Trigger breakpoint:

1. In the editor or **Disassembly** window, right-click and choose **Trace Start Trigger** from the context menu.
   Alternatively, open the **Breakpoints** window by choosing **View>Breakpoints**.
2. In the **Breakpoints** window, right-click and choose **New Breakpoint>Trace Start Trigger**.
   Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.
3. In the **Break at** text box, specify an expression, an absolute address, or a source location. Click **OK**.

4. When the breakpoint is triggered, the trace data collection starts.

Requirements

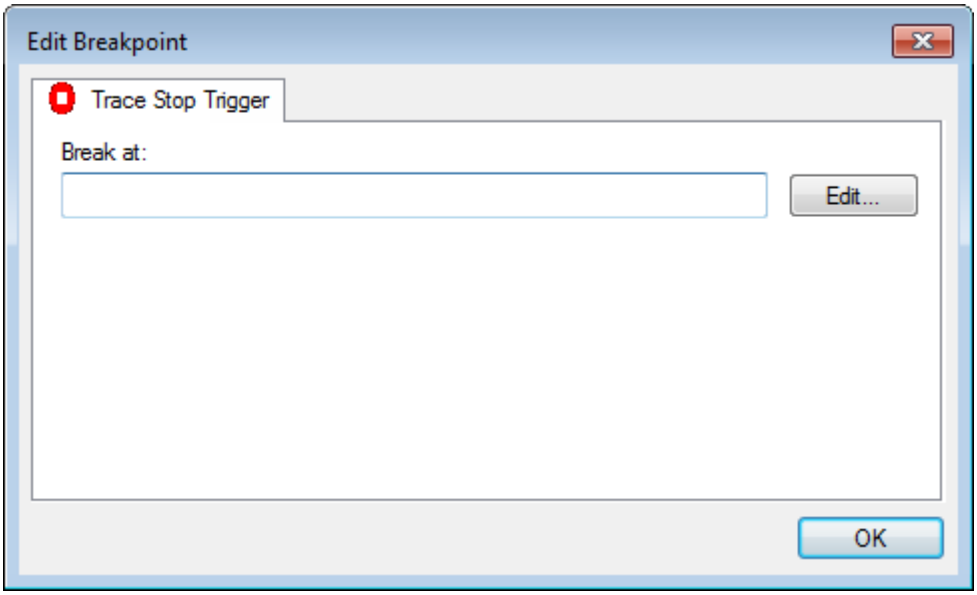
The C-SPY simulator. Not available for all cores and devices.

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 117*.

Trace Stop Trigger breakpoint dialog box

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



Use this dialog box to set a Trace Stop Trigger breakpoint where you want to stop collecting trace data. If you want to collect trace data only for a specific range, you might also need to set a Trace Start Trigger breakpoint where you want to start collecting data.

See also *Trace Start Trigger breakpoint dialog box, page 179* and *Trace data collection using breakpoints, page 160*.

To set a Trace Stop Trigger breakpoint:

1. In the editor or **Disassembly** window, right-click and choose **Trace Stop Trigger** from the context menu.
   Alternatively, open the **Breakpoints** window by choosing **View>Breakpoints**.
2. In the **Breakpoints** window, right-click and choose **New Breakpoint>Trace Stop Trigger**.
   Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.
3. In the **Break at** text box, specify an expression, an absolute address, or a source location. Click **OK**.
4. When the breakpoint is triggered, the trace data collection stops.

Requirements

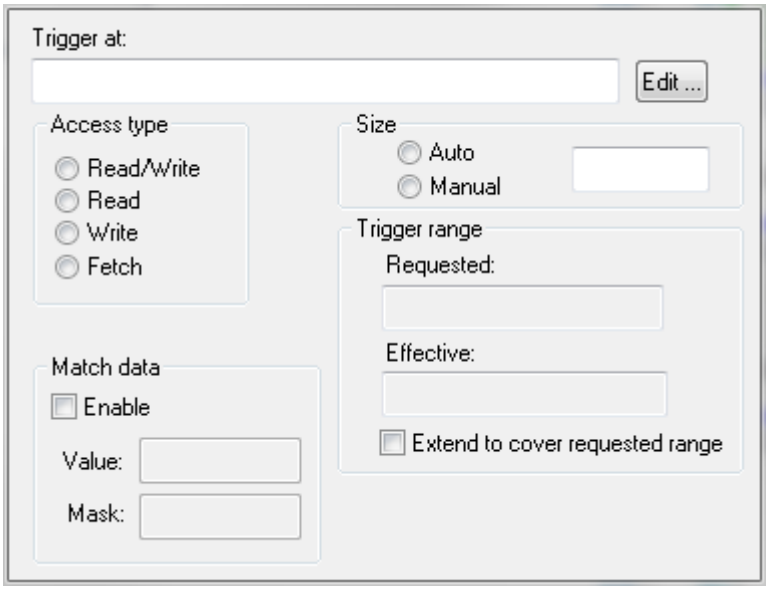
The C-SPY simulator. Not available for all cores and devices.

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 117*.

Trace Start breakpoints dialog box (I-jet 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 I-jet driver
- The C-SPY CMSIS-DAP driver
- A device with data matching capabilities.

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.



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.

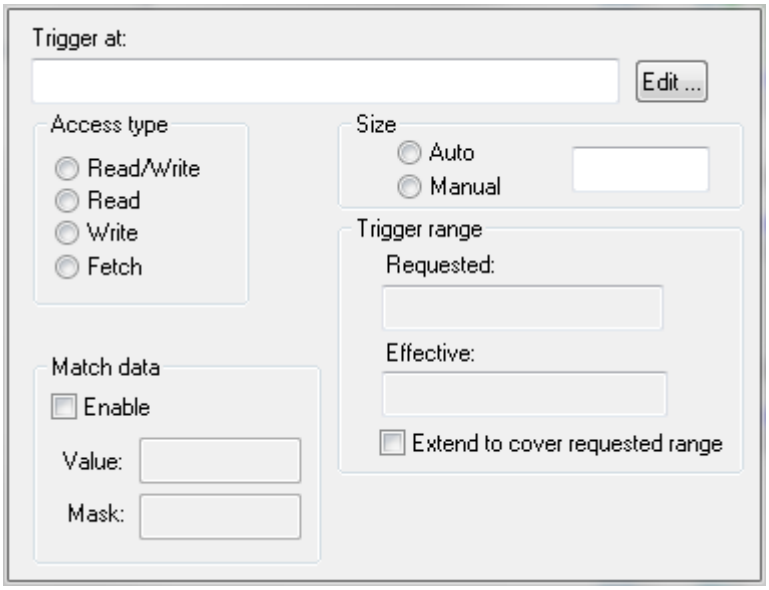
Trigger range

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 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 I-jet driver
- The C-SPY CMSIS-DAP 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.



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 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.

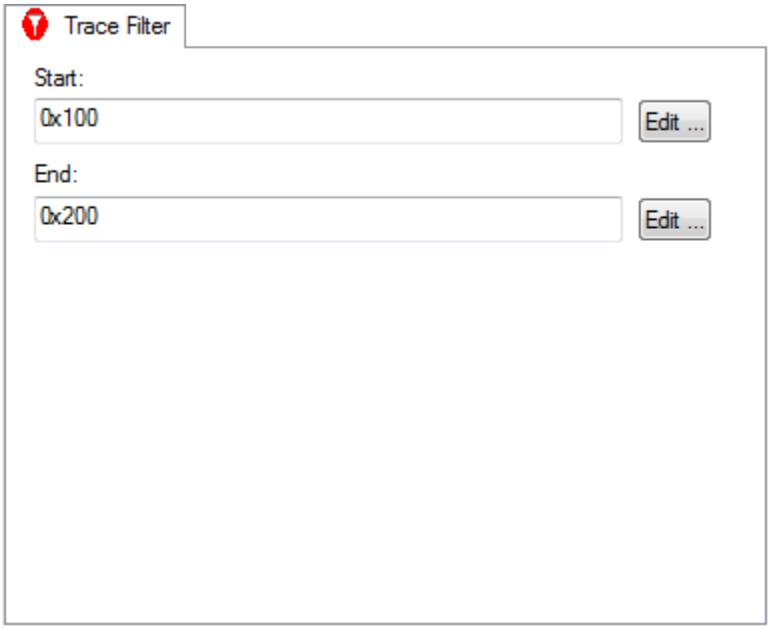
Trigger range

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.

Trace Filter breakpoints dialog box (I-jet)

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 I-jet driver
- The C-SPY CMSIS-DAP driver

Start

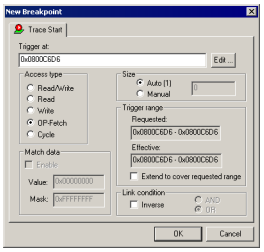
Specify the start location of the code section for which you want to collect trace data. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box, page 117*.

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 117*.

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.



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

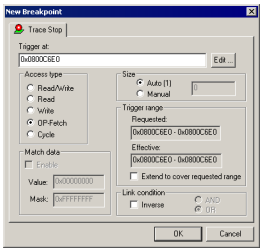
Link condition

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.



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

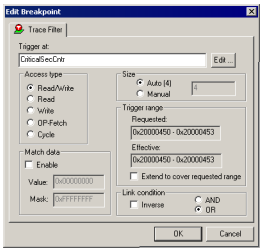
Link condition

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 117*.

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.



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

Link condition

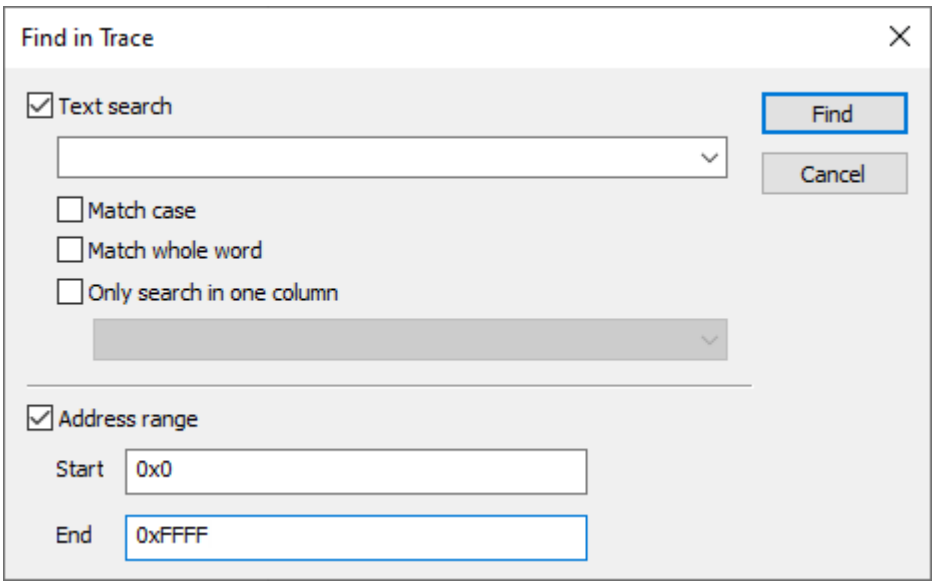
Specifies how trace conditions are combined, using **AND** and **OR**. When combining a condition that has the link condition **AND** with a condition that has the link condition **OR**, **AND** has precedence. The option **Inverse** inverts the trace condition and is individual for each trace filter 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.

Find in Trace dialog box

The **Find in Trace** dialog box is available by clicking the **Find** button on the **Trace** window toolbar or by choosing **Edit>Find and Replace>Find**.

Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the **Trace** window is the current window or the **Find** dialog box if the editor window is the current window.



Use this dialog box to specify the search criteria for advanced searches in the trace data.

The search results are displayed in the **Find in Trace** window—available from the **View>Messages** menu, see *Find in Trace window, page 193*.

See also *Searching in trace data, page 160*.

Requirements

One of these alternatives:

- The C-SPY Simulator. Not available for all cores and devices.
- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY ST-LINK driver
- The C-SPY TI XDS driver

Text search

Specify the string you want to search for. To specify the search criteria, choose between:

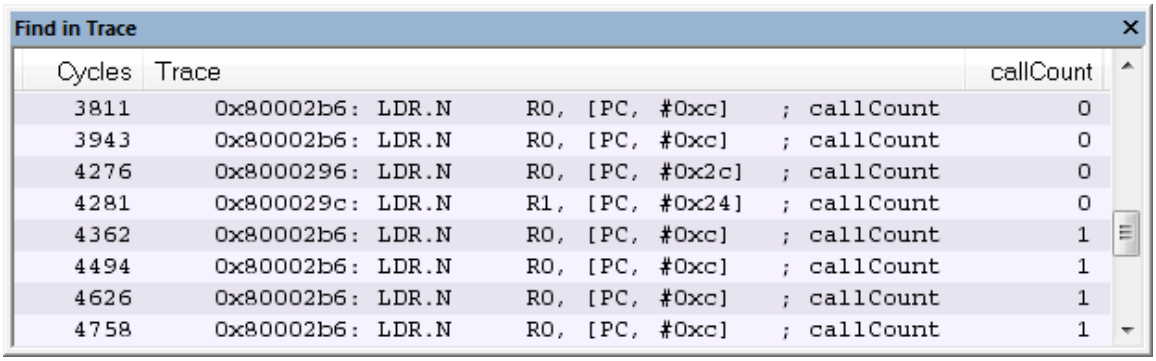
Match case	Searches only for occurrences that exactly match the case of the specified text. Otherwise **int** will also find **INT**, **Int**, and so on.
Match whole word	Searches only for the string when it occurs as a separate word. Otherwise **int** will also find **print**, **sprintf**, and so on.
Only search in one column	Searches only in the column you selected from the drop-down list.

Address range

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

Find in Trace window

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



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 191*.

See also *Searching in trace data, page 160*.

Requirements

One of these alternatives:

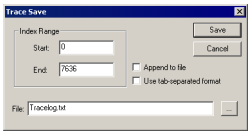
- The C-SPY Simulator. Not available for all cores and devices.
- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY ST-LINK driver

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
- The C-SPY TI XDS 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.

The application timeline

Contents

Introduction to analyzing your application's timeline 195
Briefly about analyzing the timeline 195
Requirements for timeline support 196
Analyzing your application's timeline 197
Displaying a graph in the Timeline window 197
Navigating in the graphs 197
Analyzing performance using the graph data 198
Getting started using data logging 198
Getting started using event logging 199
Reference information on application timeline 200
Data Log window 201
Data Log Summary window 203
Event Log window 205
Event Log Summary window 207
Timeline window—Call Stack graph 211
Timeline window—Data Log graph 215
Timeline window—Events graph 218
Viewing Range dialog box 221

INTRODUCTION TO ANALYZING YOUR APPLICATION'S TIMELINE

See also:

- *Trace, page 155*

Briefly about analyzing the timeline

C-SPY can provide information for various aspects of your application, collected when the application is running. This can help you to analyze the application's behavior.

You can view the timeline information in different representations:

- As different *graphs* that correlate with the running application in relation to a shared *time axis*.
- As detailed logs
- As summaries of the logs.

Depending on the capabilities of your hardware, the debug probe, and the C-SPY driver you are using, timeline information can be provided for:

Call stack	Can be represented in the **Timeline** window, as a graph that displays the sequence of function calls and returns collected by the trace system. You get timing information between the function invocations. Note that there is also a related **Call Stack** window and a **Function Trace** window, see *Call Stack window, page 59* and *Function Trace window, page 178*, respectively.
Data logging	Based on data logs collected by the trace system for up to four different variables or address ranges, specified by means of *Data Log breakpoints*. Choose to display the data logs:

- In the **Timeline** window, as a graph of how the values change over time.
- In the **Data Log** window and the **Data Log Summary** window.

Event logging

Based on *event logs* produced when the execution passes specific positions in your application code. Choose to display the event logs:

- In the **Timeline** window, as a graph of the timing of the events.
- In the **Event Log** window and the **Event Log Summary** window.

Event logs can help you to analyze the application flow and inspect data correlated to a certain position in your application code.

Interrupt logging

Based on interrupt logs collected by the trace system. Choose to display the interrupt logs:

- In the **Timeline** window, as a graph of the interrupt events during the execution of your application.
- In the **Interrupt Log** window and the **Interrupt Log Summary** window.

Interrupt logging can, for example, help you locate which interrupts you can fine-tune to make your application more efficient.

For more information, see the chapter *Interrupts, page 306*.

Power logging

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

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

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

For more information, see the chapter *Power debugging, page 243*.

State logging

Based on logged activity—state changes—for peripheral units and clocks, as well as for CPU modes generated by the debug probe or associated hardware. Choose to display the state logs:

- In the **Timeline** window, as a graph of the state changes.
- In the **State Log** window and in the **State Log Summary** window.

The information is useful for tracing the activity on the target system.

For more information, see the chapter *Power debugging, page 243*.

Requirements for timeline support

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

Target system	Call Stack	Data logging	State logging	Event logging	Interrupt logging	Power logging
C-SPY simulator	Yes[3]	Yes[3]	—	—	Yes[3]	—
CMSIS-DAP	Yes[2]	—	—	—	—	—
I-jet	Yes[2]	Yes	—	Yes	Yes	Yes
J-Link	Yes[2]	Yes	—	Yes	Yes	Yes
J-Trace	Yes	Yes[1]	—	Yes[1]	Yes[1]	—
ST-LINK	—	Yes[3]	—	Yes[3]	Yes[3]	Yes[3]
TI MSP-FET	—	—	Yes	—	—	Yes
TI XDS	—	Yes	Yes	Yes	Yes	Yes

Table 11. Support for timeline information

1 Very limited when ETM trace is enabled.

2 Requires ETB/MTB.

3 Not for all cores and devices.

For more information about requirements related to trace data, see *Requirements for using trace, page 157*.

ANALYZING YOUR APPLICATION'S TIMELINE

See also:

- *Debugging in the power domain, page 247*
- *Using the interrupt system, page 309*

Displaying a graph in the Timeline window

The **Timeline** window can display several graphs—follow this example procedure to display any of these graphs. For an overview of the graphs and what they display, see *Briefly about analyzing the timeline, page 195*.

1. Choose ***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**in 32-bit mode**, choose **Simulator>Simulated Frequency** to set up a frequency that matches the simulated hardware.
2. Choose **Timeline** from the C-SPY driver menu to open the **Timeline** window.
3. In the **Timeline** window, right-click in the window and choose **Select Graphs** from the context menu to select which graphs to be displayed.
4. In the **Timeline** window, right-click in the graph area and choose **Enable** from the context menu to enable a specific graph.
5. For the Data Log graph, you must set a Data Log breakpoint for each variable you want a graphical representation of in the **Timeline** window. See *Data Log breakpoints dialog box, page 110* and *Data Log breakpoints dialog box (C-SPY hardware drivers), page 111*, respectively.
6. 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 199*.
7. Click **Go** on the toolbar to start executing your application. The graphs that you have enabled appear.

Navigating in the graphs

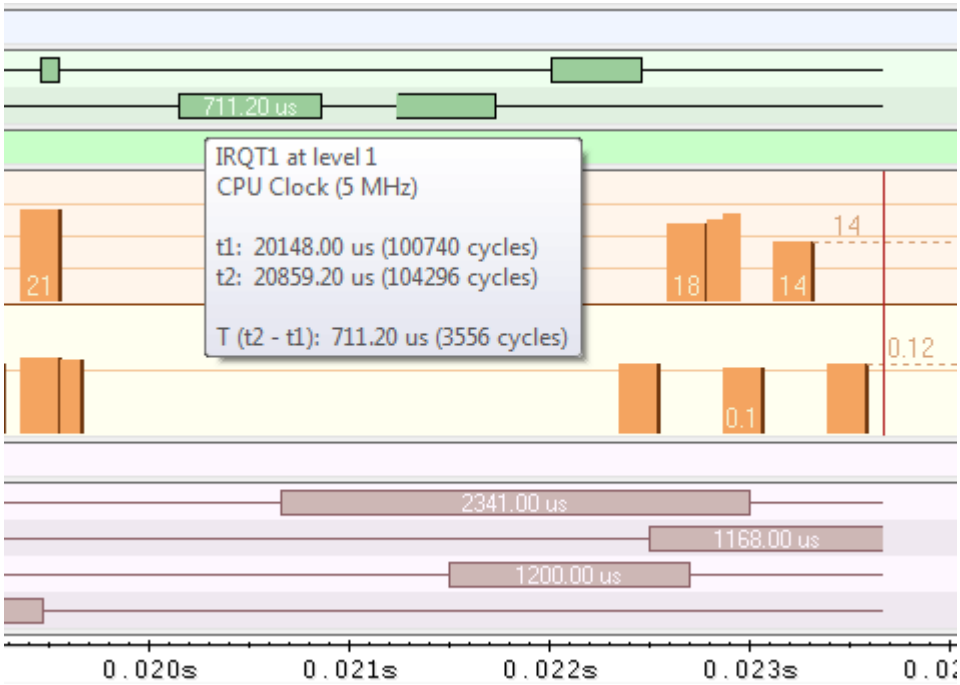
After you have performed the steps in *Displaying a graph in the Timeline window, page 197*, you can use any of these alternatives to navigate in the graph:

- Right-click and from the context menu choose **Zoom In** or **Zoom Out**. Alternatively, use the + and – keys. The graph zooms in or out depending on which command you used.
- Right-click in the graph and from the context menu choose **Navigate** and the appropriate command to move backwards and forwards on the graph. Alternatively, use any of the shortcut keys—arrow keys, Home, End, and Ctrl+End.
- Double-click on a sample of interest to highlight the corresponding source code in the editor window and in the **Disassembly** window.
- Click on the graph and drag to select a time interval, which will correlate to the running application. The selection extends vertically over all graphs, but appears highlighted in a darker color for the selected graph. Press Enter or right-click and from the context menu choose **Zoom>Zoom to Selection**. The selection zooms in. Use the navigation keys in combination with the Shift key to extend the selection.

Analyzing performance using the graph data

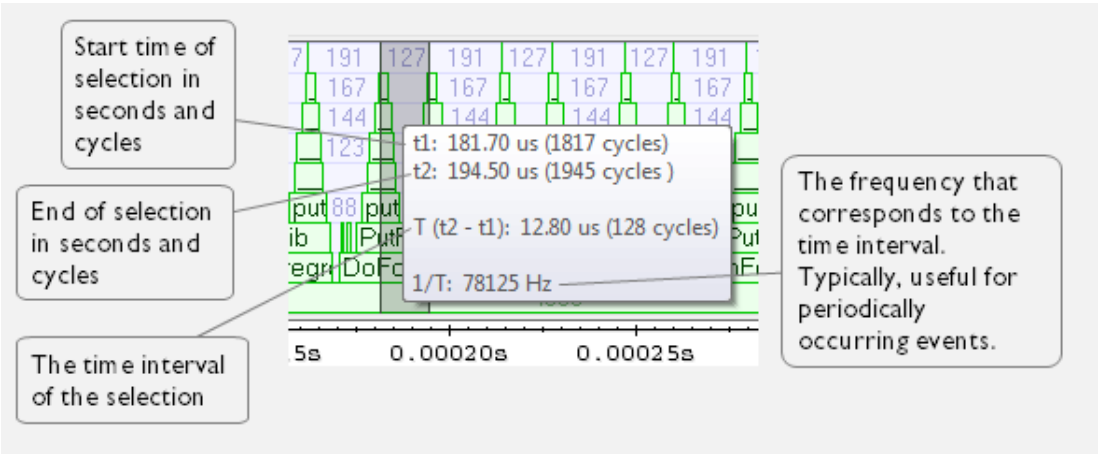
The **Timeline** window provides a set of tools for analyzing the graph data.

1. In the **Timeline** window, right-click and choose **Time Axis Unit** from the context menu. Select which unit to be used on the time axis—choose between **Seconds** and **Cycles**. If **Cycles** is not available, the graphs are based on different clock sources.
2. Execute your application to display a graph, following the steps described in *Displaying a graph in the Timeline window, page 197*.
3. Whenever execution stops, point at the graph with the mouse pointer to get detailed tooltip information for that location.



Note that if you have enabled several graphs, you can move the mouse pointer over the different graphs to get graph-specific information.

4. Click in the graph and drag to select a time interval. Point in the graph with the mouse pointer to get timing information for the selection.



Getting started using data logging

1. To set up for data logging, choose ***Driver*>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, see *Data Log breakpoints, page 92*.
3. Choose ***Driver*>Data Log** to open the **Data Log** window. Optionally, you can also choose:
   - ***Driver*>Data Log Summary** to open the **Data Log Summary** window
   - ***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 address
   - Data value + exact address

   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 the **Data Log** window, the **Data Log Summary** window, or the Data Log 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 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 ***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). See also *Timeline window—Events graph, page 218*.

- Choose *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. See also *Event Log window, page 205*.
- Choose *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. See also *Event Log Summary window, page 207*.



Whenever the Events graph or the **Event Log** window is enabled, you can also enable 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. Select the graph and right-click to view the context menu. Here you can choose to:
   - Change the radix (you can choose between displaying values in hexadecimal or in decimal format). Note that this setting affects also the **Event Log** window and the **Event Log Summary** window.
   - Show the numerical value of the variables
   - Show the value of the events
   - Select the style of the graph (as bars, levels, or linear)
   - Select the size of the graph (S, M, or L)
   - Go to source
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.

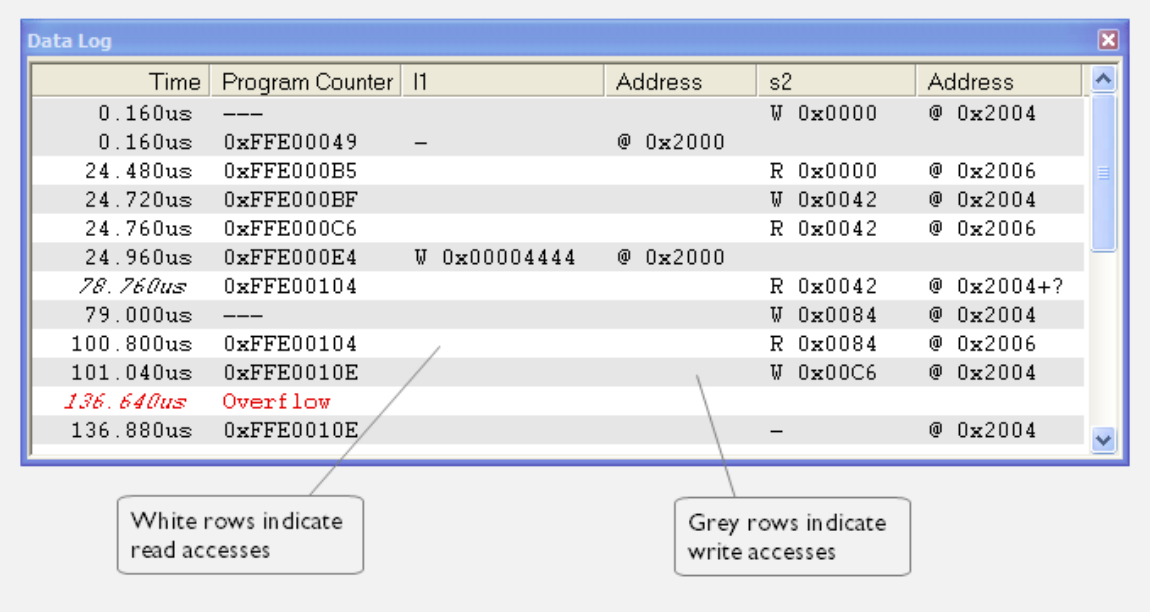
REFERENCE INFORMATION ON APPLICATION TIMELINE

See also:

- *Timeline window—Interrupt Log graph, page 324*
- *Timeline window—Power graph, page 254*
- *Timeline window—State Log graph, page 259*

Data Log window

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



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

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

See also *Getting started using data logging, page 198*.

Requirements

One of these alternatives:

- The C-SPY simulator. (Not for all cores and devices.)
- A device and a debug probe with support for SWO.

Display area

Each row in the display area shows the time, the program counter, and, for every tracked data object, its value and address. All information is cleared on reset. The information is displayed in these columns:

Time
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, the C-SPY ST-LINK driver, and the C-SPY TI XDS 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.

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`.

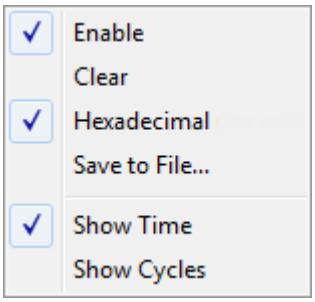
To specify what data you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints, page 92*.

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 driver, the C-SPY J-Link driver, the C-SPY ST-LINK driver, and the C-SPY TI XDS 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

This context menu is available:

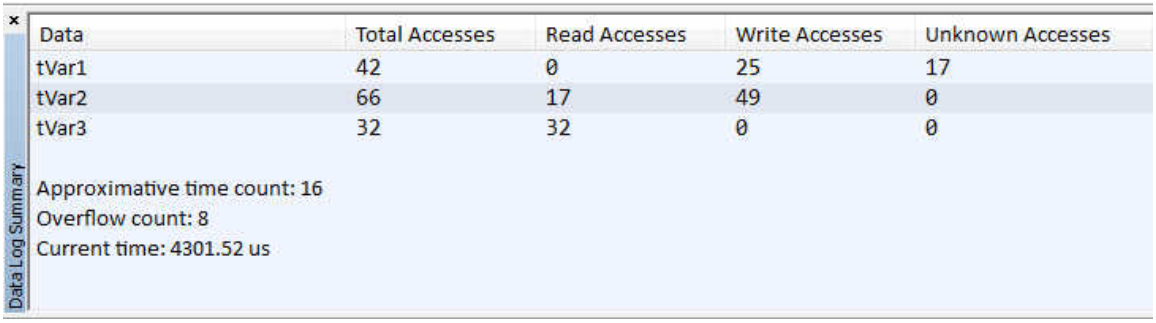


These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Hexadecimal	Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.
Save to File	Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Data Log Summary window

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



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

See also *Getting started using data logging, page 198*.

Requirements

One of these alternatives:

- The C-SPY simulator. (Not for all cores and devices.)
- A device and a debug probe with support for SWO.

Display area

Each row in this area displays the type and the number of accesses to each memory location or area in these columns. Summary information is listed at the bottom of the display area.

Data
The name of the data object you have selected to log accesses to. To specify what data object you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints, page 92*.

Total Accesses
The total number of accesses.

If the sum of read accesses and write accesses is less than the total accesses, the target system for some reason did not provide valid access type information for all accesses.

Read Accesses
The total number of read accesses.

Write Accesses
The total number of write accesses.

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

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

For some C-SPY drivers, this information is not displayed or the value is always zero. In this case, all logs have an exact time stamp.

For other C-SPY drivers, a non-zero value is displayed. The value represents the amount of logs with an approximative time stamp. This might happen if the bandwidth in the communication channel is too low compared to the amount of data packets generated by the CPU or if the CPU generated packets with an approximative time stamp.

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

For some C-SPY drivers, this information is not displayed or the value is always zero.

For other C-SPY drivers, the number represents the amount of overflows in the communication channel which can cause logs to be lost. If this happens, it indicates that logs might be incomplete. To solve this, make sure not to use all C-SPY log features simultaneously or check used bandwidth for the communication channel.

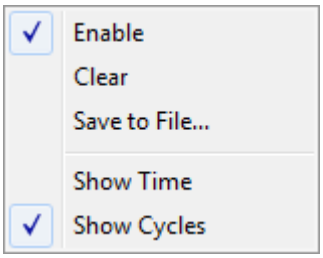
Current time/Current cycles
The information displayed depends on the C-SPY driver you are using.

For some C-SPY drivers, the value is always zero or not visible at all.

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

Context menu

This context menu is available:

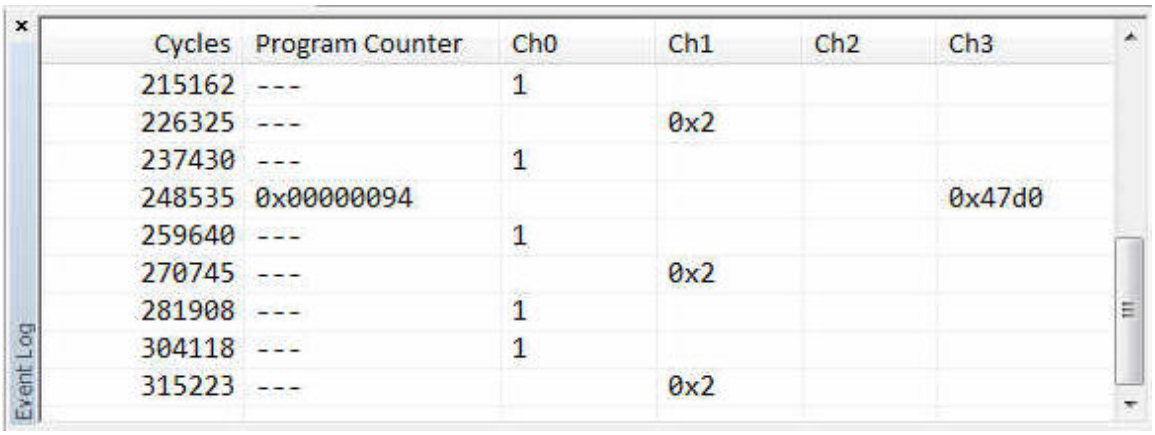


These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Save to File	Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

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 the 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.

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

See also *Getting started using event logging, page 199*.

Requirements

A Cortex device and one of these alternatives:

- The C-SPY I-jet 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
- The C-SPY ST-LINK driver and an ST-LINK debug probe with an SWD interface between the debug probe and the target system
- The C-SPY TI XDS driver and a TI XDS debug probe with an SWD interface between the debug probe and the target system.

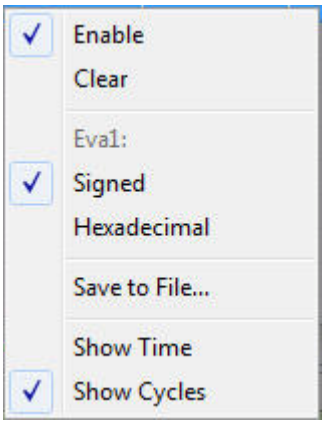
Display area

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 199*.

Context menu

This context menu is available:

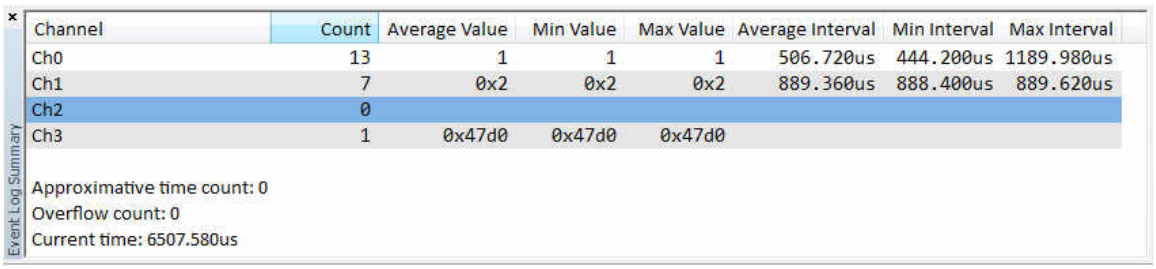


These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Variable	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).
Signed	Toggles between displaying the selected value as a signed or unsigned number. Note that this setting also affects the log window.
Hexadecimal	Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.
Save to File	Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Event Log Summary window

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



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.

See also *Getting started using event logging, page 199*.

Requirements

A Cortex device and one of these alternatives:

- The C-SPY I-jet 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
- The C-SPY ST-LINK driver and an ST-LINK debug probe with an SWD interface between the debug probe and the target system
- The C-SPY TI XDS driver and a TI XDS debug probe with an SWD interface between the debug probe and the target system.

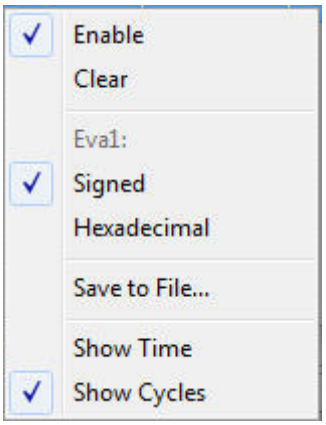
Display area

Each row displays the type and the number of accesses to each location in your application code in these columns. 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/Current cycles	The information displayed depends on the C-SPY driver you are using. For some C-SPY drivers, the value is always zero or not visible at all. For other C-SPY drivers, the number represents the current time or cycles—the number of cycles or the execution time since the start of execution.

Context menu

This context menu is available:

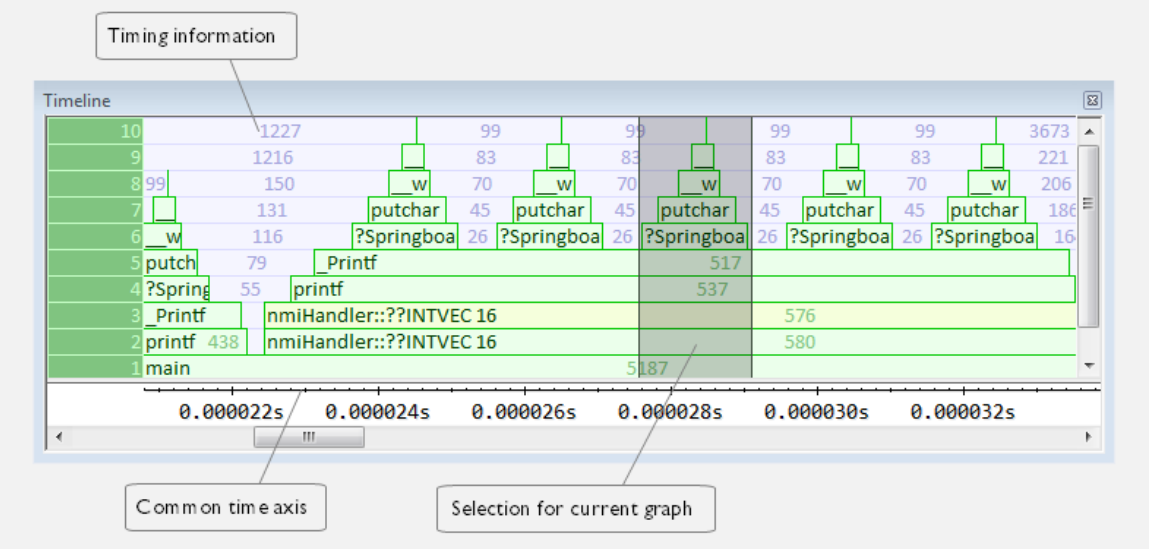


These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Variable	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).
Signed	Toggles between displaying the selected value as a signed or unsigned number. Note that this setting also affects the log window.
Hexadecimal	Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.
Save to File	Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Timeline window—Call Stack graph

The **Timeline** window is available from the *Driver* menu during a debug session.



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

The Call Stack graph displays the sequence of function calls and returns collected by the trace system.

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

Requirements

One of these alternatives:

- The C-SPY simulator. (Not for all cores and devices.)
- ETB/ETM and one of the C-SPY CMSIS-DAP driver, the C-SPY I-jet driver, or the C-SPY J-Link/J-Trace driver
- The C-SPY I-jet driver and an I-jet Trace debug probe
- The C-SPY J-Link/J-Trace driver and a J-Trace debug probe

Display area for the Call Stack graph

Each function invocation is displayed as a horizontal bar which extends from the time of entry until the return. Called functions are displayed above its caller. The horizontal bars use four different colors:

- Medium green for normal C functions with debug information
- Light green for functions known to the debugger through an assembler label
- Medium yellow for normal interrupt handlers, with debug information
- Light yellow for interrupt handlers known to the debugger through an assembler label

The timing information represents the number of cycles spent in, or between, the function invocations.

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

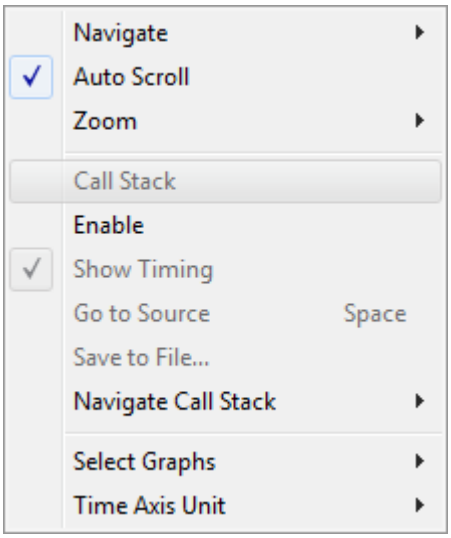
Click in the graph to display the corresponding source code.



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

Context menu

This context menu is available:



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

These commands are available:

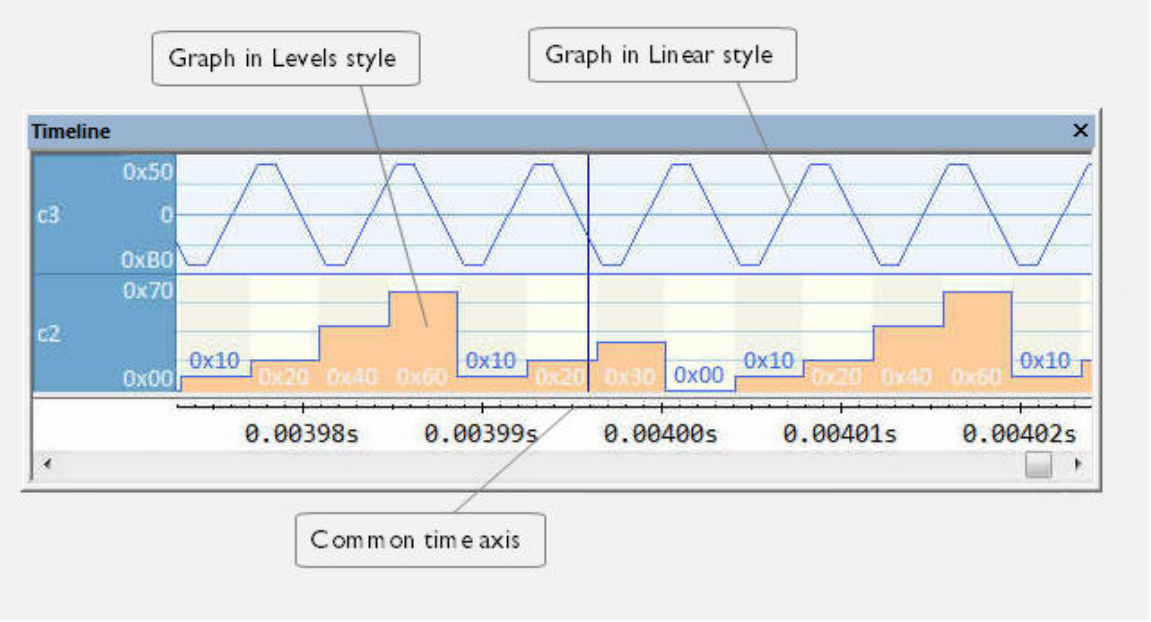
Navigate	Commands for navigating the graph(s). Choose between: **Next** moves the selection to the next relevant point in the graph. Shortcut key: right arrow. **Previous** moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow. **First** moves the selection to the first data entry in the graph. Shortcut key: Home. **Last** moves the selection to the last data entry in the graph. Shortcut key: End. **End** moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.
Auto Scroll	Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.
Zoom	Commands for zooming the window, in other words, changing the time scale. Choose between: **Zoom to Selection** makes the current selection fit the window. Shortcut key: Return. **Zoom In** zooms in on the time scale. Shortcut key: + **Zoom Out** zooms out on the time scale. Shortcut key: – **10ns**, **100ns**, **1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window. **1ms**, **10ms**, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window. **10m**, **1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.
Call Stack	A heading that shows that the Call stack-specific commands below are available.

Enable	Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.
Show Timing	Toggles the display of the timing information on or off.
Go To Source	Displays the corresponding source code in an editor window, if applicable.
Save to File	Saves all contents (or the selected contents) of the Call Stack graph to a file. The menu command is only available when C-SPY is not running.
Navigate Call Stack>After Current Loop (L)	Identifies the selected program counter and scans the trace data forward, collecting program counters, until it finds the same address again. It has now detected a loop. (Loops longer than 1000 instructions are not detected.) Then it navigates forward until it finds a program counter that is not part of the collected set. This is useful for navigating out of many iterations of an idle or polling loop.
Navigate Call Stack>Before Current Loop (Shift+L)	Behaves as **After Current Loop**, but navigates backward out of the loop.
Navigate Call Stack>After Current Function (U)	Navigates to the next unmatched return instruction. This is similar to stepping out of the current function.
Navigate Call Stack>Before Current Function (Shift+U)	Navigates to the closest previous unmatched call instruction.
Navigate Call Stack>Next Statement (S)	Navigates to the next instruction that belongs to a different C statement than the starting point. It skips function calls, i.e. it tries to reach the next statement in the starting frame.
Navigate Call Stack>Previous Statement (Shift+S)	Behaves as **Next statement**, but navigates backward to the closest previous different C statement.
Navigate Call Stack>Next on Same Address (A)	Navigates to the next instance of the starting program counter address, typically to the next iteration of a loop.
Navigate Call Stack>Previous on Same Address (Shift+A)	Navigates to the closest previous instance of the starting program counter address.
Navigate Call Stack>Next Interrupt (I)	Navigates to the next interrupt entry. (To then find the matching interrupt exit, follow up with **After Current Function**.)
Navigate Call Stack>Previous Interrupt (Shift+I)	Navigates to the closest previous interrupt entry.
Navigate Call Stack>Next Execution Start Point (E)	Navigates to the next point where the CPU was started, for example places where the application stopped at breakpoints, or was stepped.

Navigate Call Stack>Previous Execution Start Point (Shift+E)	Navigates to the closest previous point where the CPU was started.
Navigate Call Stack>Next Discontinuity (D)	Navigates to the next discontinuity in the trace data.
Navigate Call Stack>Previous Discontinuity (Shift+D)	Navigates to the closest previous discontinuity in the trace data.
Select Graphs	Selects which graphs to be displayed in the **Timeline** window.
Time Axis Unit	Selects the unit used in the time axis—choose between **Seconds** and **Cycles**. If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.
Profile Selection	Enables profiling time intervals in the **Function Profiler** window. Note that this command is only available if the C-SPY driver supports PC Sampling. See *Selecting a time interval for profiling information, page 228*.

Timeline window—Data Log graph

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



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

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

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

Requirements

One of these alternatives:

- The C-SPY simulator. (Not for all cores and devices.)
- A device and a debug probe with support for SWO.

Display area for the Data Log graph

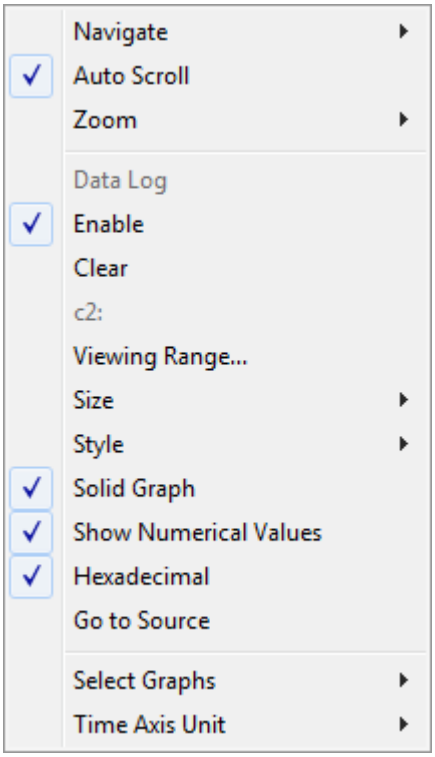
Where:

- The label area at the left end of the graph displays the variable name or the address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the Y-axis for a variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the **Data Log** window, see *Data Log window, page 201*.
- The graph can be displayed either as a thin line between consecutive logs or as a rectangle for every log (optionally color-filled).
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system. A red question mark indicates a log without a value.

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

Context menu

This context menu is available:



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

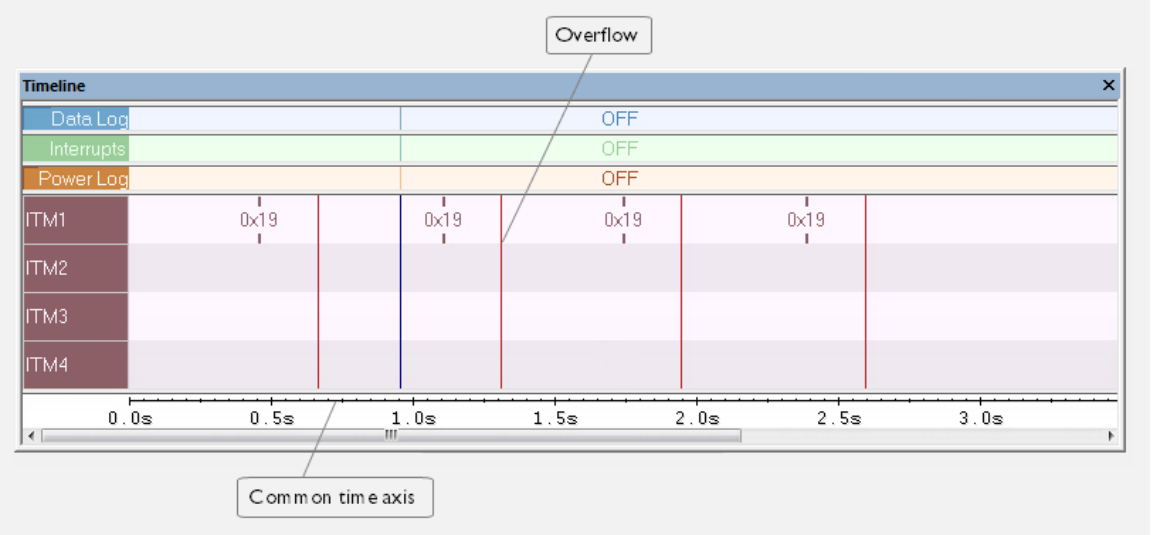
These commands are available:

Navigate	Commands for navigating the graph(s). Choose between: **Next** moves the selection to the next relevant point in the graph. Shortcut key: right arrow. **Previous** moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow. **First** moves the selection to the first data entry in the graph. Shortcut key: Home. **Last** moves the selection to the last data entry in the graph. Shortcut key: End. **End** moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.
Auto Scroll	Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.
Zoom	Commands for zooming the window, in other words, changing the time scale. Choose between: **Zoom to Selection** makes the current selection fit the window. Shortcut key: Return. **Zoom In** zooms in on the time scale. Shortcut key: + **Zoom Out** zooms out on the time scale. Shortcut key: – **10ns**, **100ns**, **1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window. **1ms**, **10ms**, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window.

	10m, **1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.
Data Log	A heading that shows that the Data Log-specific commands below are available.
Enable	Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Variable	The name of the variable for which the Data Log-specific commands below apply. This menu command is context-sensitive, which means it reflects the Data Log graph you selected in the **Timeline** window (one of up to four).
Viewing Range	Displays a dialog box, see *Viewing Range dialog box, page 221*.
Size	Determines the vertical size of the graph—choose between **Small**, **Medium**, and **Large**.
Style	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.
Solid Graph	Displays the graph as a color-filled solid graph instead of as a thin line.
Show Numerical Value	Shows the numerical value of the variable, in addition to the graph.
Hexadecimal	Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.
Go To Source	Displays the corresponding source code in an editor window, if applicable.
Select Graphs	Selects which graphs to be displayed in the **Timeline** window.
Time Axis Unit	Selects the unit used in the time axis—choose between **Seconds** and **Cycles**. If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.

Timeline window—Events graph

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



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

The Events graph displays the events produced when the execution passes specific positions in your application code.

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

Requirements

One of these alternatives:

- The C-SPY I-jet driver and an I-jet debug probe
- The C-SPY J-Link/J-Trace driver. For the J-Trace debug probe, event logging is very limited when ETM trace is enabled.
- The C-SPY ST-LINK driver
- The C-SPY TI XDS driver

Display area for the Events graph

Where:

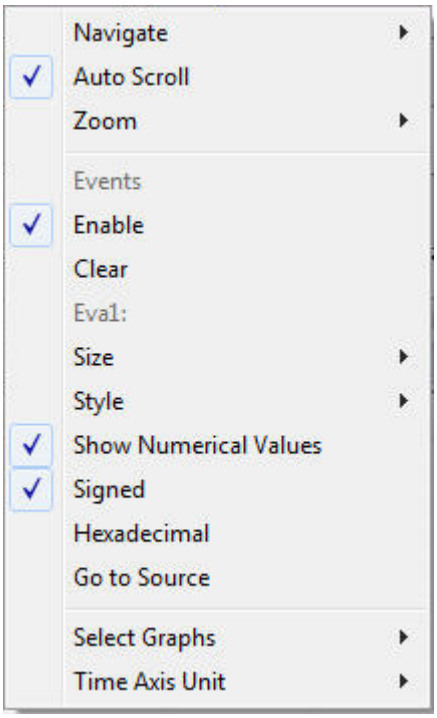
- The label area at the left end of the graph displays the name of the channel.
- For each channel, there will be a vertical line that indicates when the event occurred. Optionally, you can choose to display the event value that was passed with the event.
- The graph can be displayed in different styles—as 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 shared time axis that uses seconds or cycles as the time unit.

See also *Getting started using event logging, page 199*.

Context menu

This context menu is available:



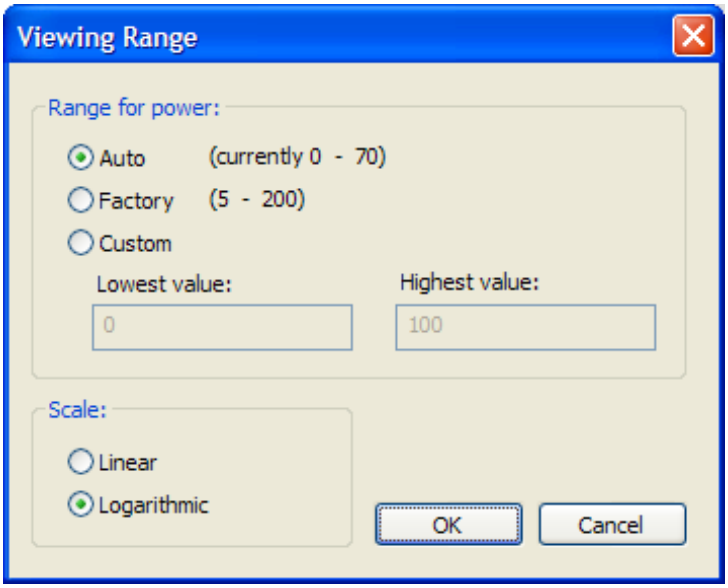
These commands are available:

Navigate	Commands for navigating the graph(s). Choose between: **Next** moves the selection to the next relevant point in the graph. Shortcut key: right arrow. **Previous** moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow. **First** moves the selection to the first data entry in the graph. Shortcut key: Home. **Last** moves the selection to the last data entry in the graph. Shortcut key: End. **End** moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.
Auto Scroll	Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.
Zoom	Commands for zooming the window, in other words, changing the time scale. Choose between: **Zoom to Selection** makes the current selection fit the window. Shortcut key: Return. **Zoom In** zooms in on the time scale. Shortcut key: + **Zoom Out** zooms out on the time scale. Shortcut key: – **10ns**, **100ns**, **1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window. **1ms**, **10ms**, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window. **10m**, **1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Events	A heading that shows that the Events-specific commands below are available.
Enable	Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Variable	The name of the 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).
Viewing Range	Displays a dialog box, see *Viewing Range dialog box, page 221*.
Size	Determines the vertical size of the graph—choose between **Small**, **Medium**, and **Large**.
Style	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	Shows the numerical value of the variable, in addition to the graph.
Signed	Toggles between displaying the selected value as a signed or unsigned number. Note that this setting also affects the log window.
Hexadecimal	Toggles between displaying the selected value in decimal or hexadecimal format. Note that this setting also affects the log window.
Go To Source	Displays the corresponding source code in an editor window, if applicable.
Select Graphs	Selects which graphs to be displayed in the **Timeline** window.
Time Axis Unit	Selects the unit used in the time axis—choose between **Seconds** and **Cycles**. If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.

Viewing Range dialog box

The **Viewing Range** dialog box is available from the context menu that appears when you right-click in any graph in the **Timeline** window that uses the linear, levels or columns style.



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

Requirements

One of these alternatives:

- The C-SPY Simulator. (Not for all cores and devices.)
- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver
- The C-SPY TI XDS 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 Power Log graph—Uses the range according to the properties of the measuring hardware (only if supported by the product edition you are using).

For all other graphs—Uses the range according to the value range of the variable, for example 0–65535 for an unsigned 16-bit integer.

Custom Use the text boxes to specify an explicit range.

Scale

Selects the scale type of the Y-axis:

- Linear
- Logarithmic

Profiling

Contents

Introduction to the profiler 224
Reasons for using the profiler 224
Briefly about the profiler 224
Requirements for using the profiler 225
Using the profiler 226
Getting started using the profiler on function level 226
Analyzing the profiling data 226
Getting started using the profiler on instruction level 228
Selecting a time interval for profiling information 228
Reference information on the profiler 229
Function Profiler window 229

INTRODUCTION TO 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.

For debug probes that support it, C-SPY can capture full instruction trace in real time, and process the information for the **Function Profiler** window.

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

Profiling sources

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

- *Trace (calls)*

The full instruction trace (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.

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.



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

Requirements for using the profiler

The C-SPY simulator supports the profiler for most cores and devices. If supported, there are no further 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 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 I-jet 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	Yes[4]	Yes[4]	—	—
CMSIS-DAP	Yes	Yes	—	—
I-jet	Yes	Yes	Yes[1]	Yes
I-jet Trace	Yes	Yes	Yes[1]	Yes
J-Link	Yes	Yes	Yes[1]	—
J-Link Ultra	Yes	Yes	Yes[1]	Yes[2]
J-Trace	Yes	Yes	Yes[1]	—
GDB Server	—	—	—	—
ST-LINK	—	—	Yes[1]	Yes[1]
TI Stellaris	—	—	—	—
TI XDS	—	—	Yes[1]	Yes[3]
TI MSP-FET	—	—	—	Yes

Table 12. C-SPY driver profiling support

1 Only for Cortex-M devices supporting SWO.

2 Requires SWO trace.

3 Only for XDS110 with EnergyTrace™ options.

4 Not for all cores and devices.

USING THE PROFILER

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

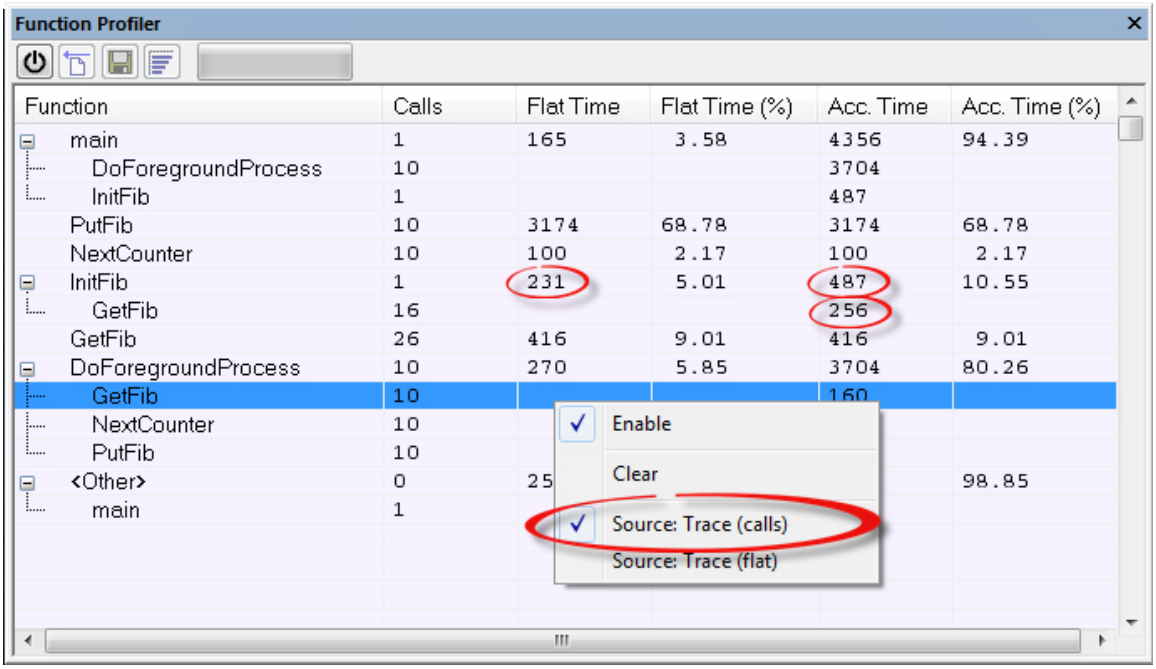
2. To set up the profiler for function profiling:
   - If you use ETM trace, make sure that the **Cycle accurate tracing** option is selected in the **Trace Settings** dialog box.
   - If you use the SWD/SWO interface, no specific settings are required.
3.  When you have built your application and started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

Analyzing the profiling data

Here follow some examples of how to analyze the data.

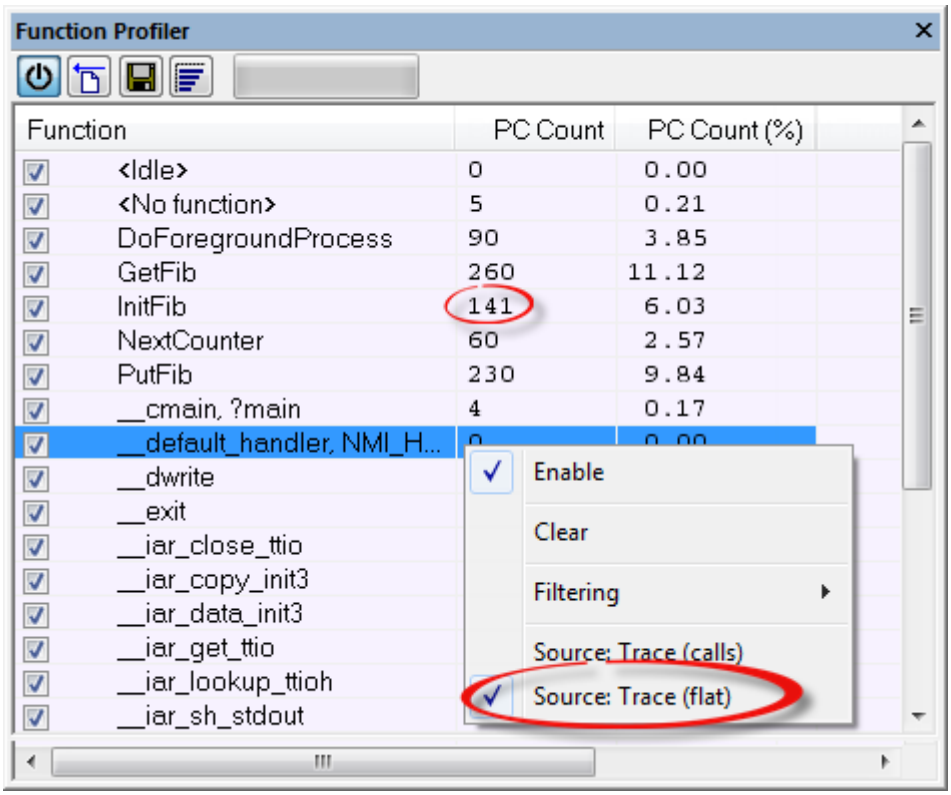
The first figure shows the result of profiling using **Source: Trace (calls)**. The profiler follows the program flow and detects function entries and exits.

- For the **InitFib** function, **Flat Time** 231 is the time spent inside the function itself.
- For the **InitFib** function, **Acc Time** 487 is the time spent inside the function itself, including all functions InitFib calls.
- For the **InitFib/GetFib** function, **Acc Time** 256 is the time spent inside **GetFib** (but only when called from **InitFib**), including any functions **GetFib** calls.
- Further down in the data, you can find the **GetFib** function separately and see all of its subfunctions (in this case none).



The second figure shows the result of profiling using **Source: Trace (flat)**. In this case, the profiler does not follow the program flow, instead the profiler only detects whether the `PC` address is within the function scope. For incomplete trace data, the data might contain minor errors.

For the **InitFib** function, **Flat Time** 231 is the time (number of hits) spent inside the function itself.



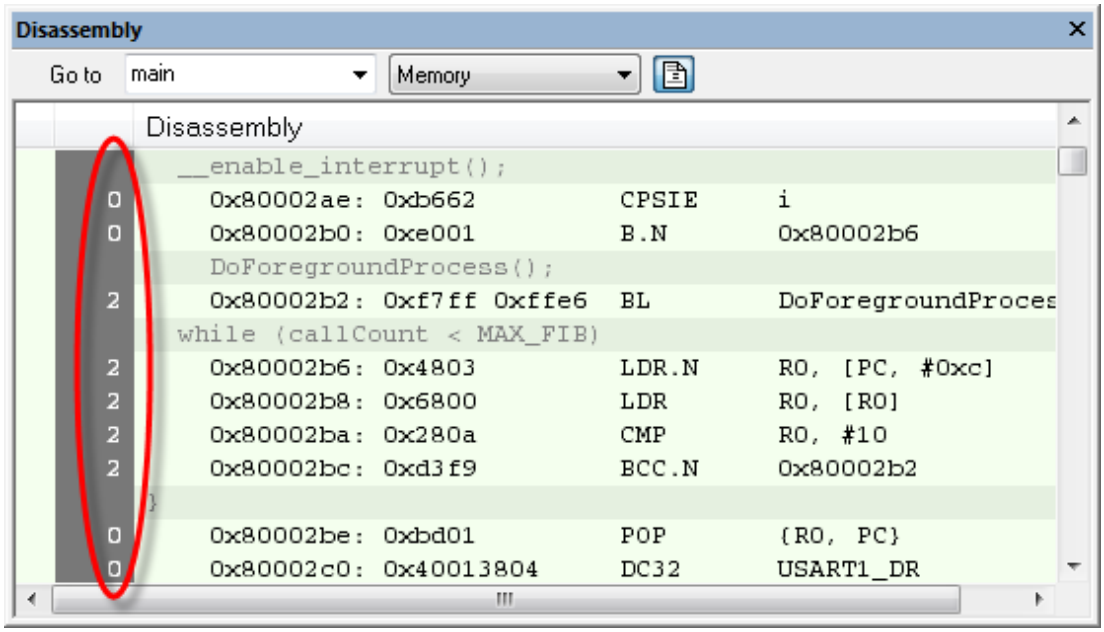
To secure valid data when using a debug probe, make sure to use the maximum trace buffer size and set a breakpoint in your code to stop the execution before the buffer is full.

The <No function> entry represents PC values that are not within the known C-SPY ranges for the application.

Getting started using the profiler on instruction level

To display instruction profiling information in the Disassembly window:

1. When you have built your application and started C-SPY, choose **View>Disassembly** to open the **Disassembly** window, and choose **Instruction Profiling>Enable** from the context menu that is available when you right-click in the left-hand margin of the **Disassembly** window.
2. Make sure that the **Show** command on the context menu is selected, to display the profiling information.
3. Start executing your application to collect the profiling information.
4. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, you can view instruction level profiling information in the left-hand margin of the window.



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

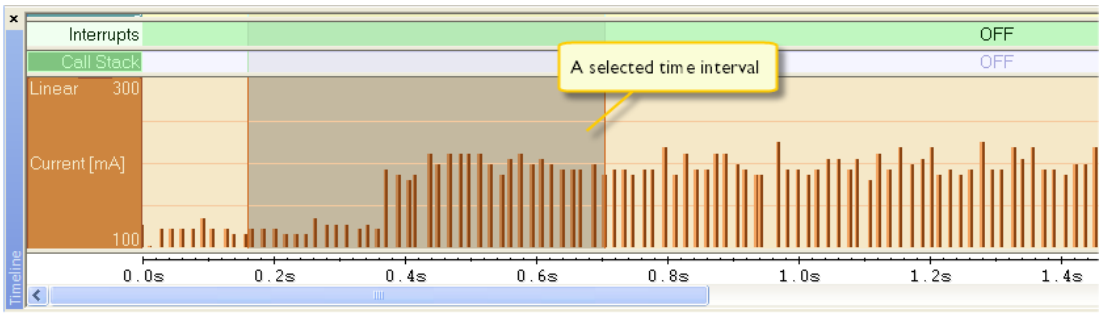
Instruction profiling attempts to use the same source as the function profiler. If the function profiler is not on, the instruction profiler will try to use first trace and then PC sampling as source. You can change the source to be used from the context menu that is available in the **Function Profiler**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

Selecting a time interval for profiling information

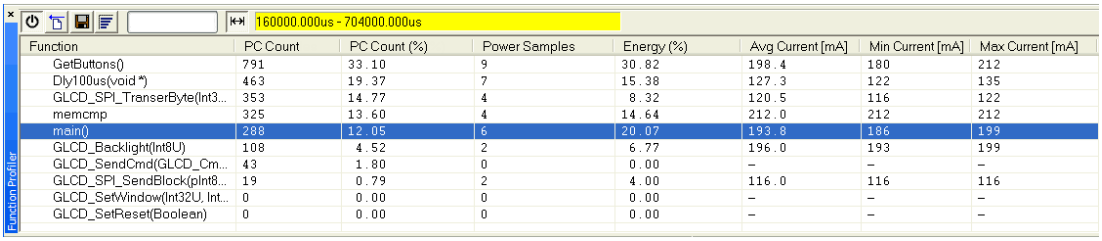
Normally, the profiler computes its information from all PC samples it receives, accumulating more and more information until you explicitly clear the profiling information. However, you can choose a time interval for which the profiler computes the PC samples. This function is supported by the I-jet and I-jet Trace in-circuit debugging probes, the J-Link probe, the J-Trace probe, the ST-LINK probe, and the TI XDS 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 ***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 Count	PC Count (%)	Power Samples	Energy (%)	Avg Current [mA]	Min Current [mA]	Max Current [mA]
GetButtons()	791	33.10	9	30.82	198.4	180	212
Dly100us(void *)	463	19.37	7	15.38	127.3	122	135
GLCD_SPI_TranserByte(Int3...	353	14.77	4	8.32	120.5	116	122
memcmp	325	13.60	4	14.64	212.0	212	212
main()	288	12.05	6	20.07	193.8	186	199
GLCD_Backlight(Int8U)	108	4.52	2	6.77	196.0	193	199
GLCD_SendCmd(GLCD_Cm...	43	1.80	0	0.00	–	–	–
GLCD_SPI_SendBlock(pInt8...	19	0.79	2	4.00	116.0	116	116
GLCD_SetWindow(Int32U, Int...	0	0.00	0	0.00	–	–	–
GLCD_SetReset(Boolean)	0	0.00	0	0.00	–	–	–

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

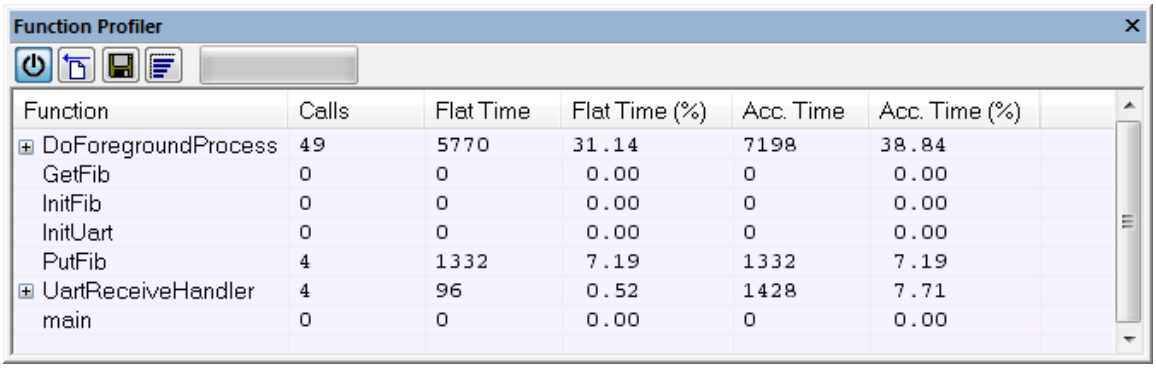
REFERENCE INFORMATION ON THE PROFILER

See also:

- *Disassembly window, page 55*
- *ETM Trace Settings dialog box (J-Link/J-Trace), page 163*
- *ETM Trace Settings dialog box (I-jet), page 161*
- *SWO Trace Window Settings dialog box, page 165*
- *SWO Configuration dialog box, page 167*

Function Profiler window

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



Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
⊞ DoForegroundProcess	49	5770	31.14	7198	38.84
GetFib	0	0	0.00	0	0.00
InitFib	0	0	0.00	0	0.00
InitUart	0	0	0.00	0	0.00
PutFib	4	1332	7.19	1332	7.19
⊞ UartReceiveHandler	4	96	0.52	1428	7.71
main	0	0	0.00	0	0.00

This window displays function profiling information.

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

See also *Using the profiler, page 226*.

Requirements

One of these alternatives:

- The C-SPY Simulator. Not available for all cores and devices.
- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY ST-LINK driver
- The C-SPY TI XDS 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 225*.

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 225*.

Display area

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

- *For the Trace (calls) source*, the display area contains one line for each function compiled with debug information enabled. When some profiling information has been collected, it is possible to expand rows of functions that have called other functions. The child items for a given function list all the functions that have been called by the parent function and the corresponding statistics.
- *For the Trace (flat) source*, the display area contains one line for each C function of your application, but also lines for sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels. Each executed `PC` address from trace data is treated as a separate sample and is associated with the corresponding line in the **Profiling** window. Each line contains a count of those samples.

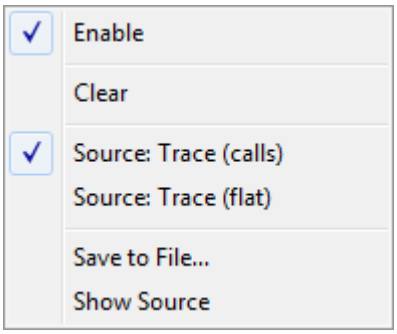
For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler, page 225*.

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

Function (All sources)	The name of the profiled C function. For Sampling source, sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels, is displayed.
Calls (Trace (calls))	The number of times the function has been called.
Flat time (Trace (calls))	The time expressed in cycles spent inside the function.
Flat time (%) (Trace (calls))	Flat time expressed as a percentage of the total time.
Acc. time (Trace (calls))	The time expressed in cycles spent inside the function and everything called by the function.
Acc. time (%) (Trace (calls))	Accumulated time expressed as a percentage of the total time.
PC Count (Trace (flat) and Sampling)	The number of executed instructions (Trace) or PC samples (Sampling) associated with the function.
PC Count (%) (Trace (flat) and Sampling)	The number of executed instructions (Trace) or PC samples (Sampling) associated with the function as a percentage of the total number of executed instructions /PC samples.
Power Samples (Power Sampling)	The number of power samples associated with that function.
Energy (%) (Power Sampling)	The accumulated value of all measurements associated with that function, expressed as a percentage of all measurements.
Avg Current [mA] (Power Sampling)	The average measured value for all samples associated with that function.
Min Current [mA] (Power Sampling)	The minimum measured value for all samples associated with that function.
Max Current [mA] (Power Sampling)	The maximum measured value for all samples associated with that function.

Context menu

This context menu is available:



The contents of this menu depend on the C-SPY driver you are using.

These commands are available:

Enable — Enables the profiler. The system will also collect information when the window is closed.

Clear — Clears all profiling data.

Filtering — Selects which part of your code to profile. Choose between:

Check All—Excludes all lines from the profiling.

Uncheck All—Includes all lines in the profiling.

Load—Reads all excluded lines from a saved file.

Save—Saves all excluded lines to a file. Typically, this can be useful if you are a group of engineers and want to share sets of exclusions.

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

Source — Selects which source to be used for the profiling information. See also *Profiling sources, page 224*.

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

Choose between:

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

Trace (calls)—the instruction count for instruction profiling is only as complete as the collected trace data.

Trace (flat)—the instruction count for instruction profiling is only as complete as the collected trace data.

Power Sampling — Toggles power sampling information on or off. This command is supported by the I-jet and I-jet Trace in-circuit debugging probes, 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.

Code coverage

Contents

Introduction to code coverage 233
Reasons for using code coverage 233
Briefly about code coverage 233
Requirements and restrictions for using code coverage 233
Reference information on code coverage 233
Code Coverage window 234

INTRODUCTION TO CODE COVERAGE

Reasons for using code coverage

The code coverage functionality is useful when you design your test procedure to verify whether all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

Briefly about code coverage

The **Code Coverage** window reports the status of the current code coverage analysis for C or C++ code. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

For debug probes that support it, C-SPY can capture full instruction trace in real time, and process the information for the **Code Coverage** window.



Assembler code is not covered in the **Code Coverage** window. To view code coverage for assembler code, use the **Disassembly** window.

Requirements and restrictions for using code coverage

Code coverage is supported by the C-SPY simulator for most cores and devices and if supported there are no specific requirements or restrictions.

To use code coverage in your hardware debugger system, consider these requirements and restrictions:

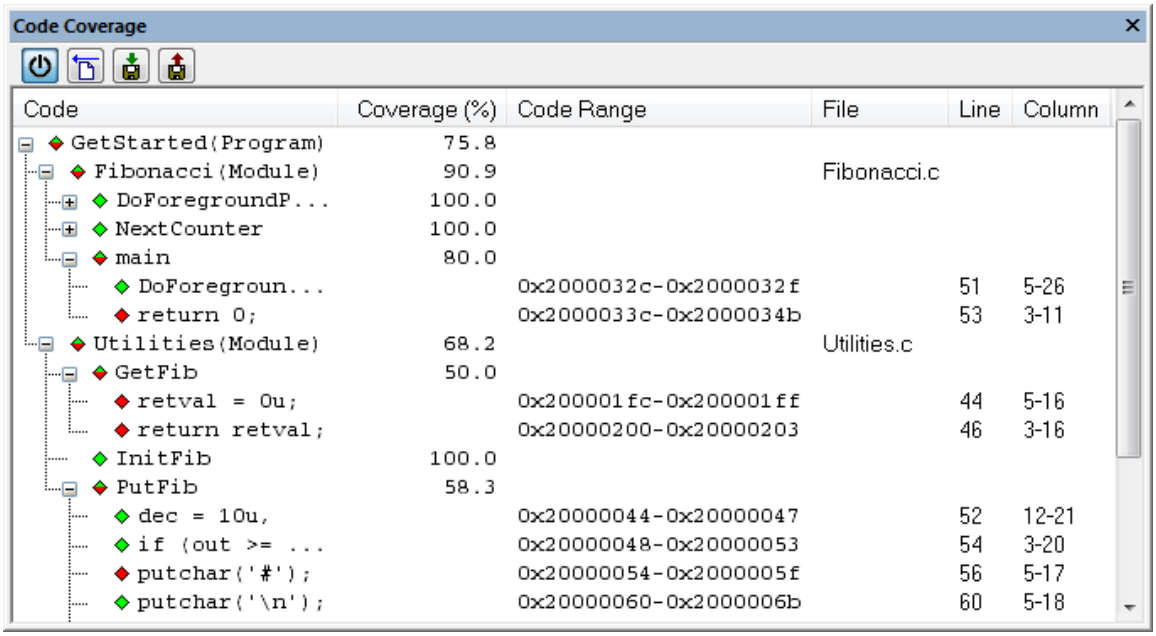
- *When SWO trace is used*—code coverage information is based on trace samples only. This means that a function must be executed several times before 100% code coverage is reached. Also, no code coverage information is collected while single stepping.
- *When ETM trace is used*—for some devices, code coverage can be of infinite length. Without this enhanced trace capability, the coverage length is restricted by 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.
- Real-time code coverage requires that the debug probe supports this feature.
- Run code coverage on non-optimized code, as optimizations will negatively affect code coverage.

REFERENCE INFORMATION ON CODE COVERAGE

See also *Single stepping, page 50*.

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.

Only source code that was compiled with debug information is displayed. Therefore, startup code, exit code, and library code are not displayed in the window. Furthermore, coverage information for statements in inlined functions is not displayed. Only the statement containing the inlined function call is marked as executed.

A statement is considered to be executed when all its instructions have been executed. By default, when a statement has been executed, it is removed from the window and the percentage is increased correspondingly.

Requirements

One of these alternatives:

- The C-SPY Simulator. Not available for all cores and devices.
- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY CMSIS-DAP driver
- The C-SPY ST-LINK driver
- The C-SPY TI XDS driver

Toolbar

The toolbar contains buttons for switching code coverage on and off, clearing the code coverage information, and saving/restoring the code coverage session. See the description of the context menu for more detailed information.

The toolbar contains these buttons:

Activate



Switches code coverage on and off during execution.

Clear



Clears the code coverage information. All step points are marked as not executed.

Save session



Saves your code coverage session data to a `*.dat` file. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command might not be supported by the C-SPY driver you are using.

Restore session



Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command might not be supported by the C-SPY driver you are using.

Display area

Double-clicking a statement or a function in the **Code Coverage** window displays that statement or function as the current position in the editor window, which becomes the active window.

These columns are available:

Code

The code coverage information is displayed as a tree structure, showing the program, module, function, and statement levels. You can use the plus (+) sign and minus (-) sign icons to expand and collapse the structure.

These icons give you an overview of the current status on all levels:

- Red diamond—0% of the modules or functions has been executed.
- Green diamond—100% of the modules or functions has been executed.
- Red and green diamond—Some of the modules or functions have been executed. This is most directly visible at the instruction level for a branch instruction or a conditionally executed instruction. For example, coverage data can indicate that while a branch instruction has executed, control flow has not both branched and continued in straight execution. Or, in the case of a conditional instruction, it indicates if the instruction has executed both with the controlling flag set, and with the controlling flag not set.

Red, green, and yellow colors can be used as highlight colors in the source editor window. In the editor window, the yellow color signifies partially executed.

Coverage (%)

The amount of statements that has been covered so far, that is, the number of executed statements divided with the total number of statements.

Code Range

The address range in code memory where the statement is located.

File

The source file where the step point is located.

Line

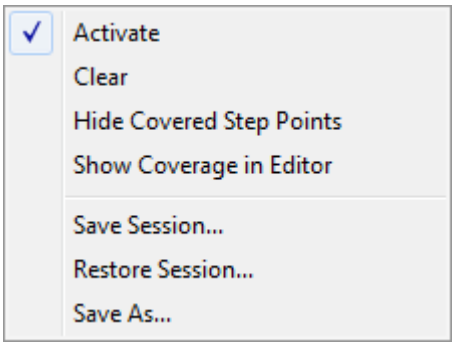
The source file line where the step point is located.

Column

The source file column where the step point is located.

Context menu

This context menu is available:



These commands are available:

Activate



Switches code coverage on and off during execution.

Clear



Clears the code coverage information. All step points are marked as not executed.

Hide Covered Step Points

Toggles the display of covered step points on and off. When this option is selected, executed statements are removed from the window.

Show Coverage in Editor

Toggles the red, green, and yellow highlight colors that indicate code coverage in the source editor window on and off.

Save Session



Saves your code coverage session data to a `*.dat` file. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command is only supported by the C-SPY simulator.

Restore Session



Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command is only supported by the C-SPY simulator.

Save As

Saves the current code coverage result in a text file.

Performance monitoring

Contents

Introduction to performance monitoring 238
Briefly about performance monitoring 238
Requirements and restrictions for using performance monitoring 238
Event types 238
Detecting counter overflow 239
Setting up performance monitoring 239
Reference information on performance monitoring 240
Performance Monitoring window 240

INTRODUCTION TO PERFORMANCE MONITORING

Briefly about performance monitoring

The **Performance Monitoring** window is a viewer for counting events or CPU clock cycles through the Performance Monitoring Unit (PMU) available on Cortex-A, Cortex-R, and some Cortex-M devices. Cortex-M devices must be based on the Armv8.1M architecture.

To monitor event counters or CPU cycles through the PMU in the **Performance Monitoring** window, you must first make settings in the window and select the events that you want to monitor.

Requirements and restrictions for using performance monitoring

To use performance monitoring in your hardware debugger system, consider these requirements and restrictions:

- This feature requires an I-jet or J-Link probe.
- The target device must be equipped with a PMU. There are different PMU protocol versions available. C-SPY can handle all versions from PMUv1 and higher.

  Note: The PMU on Arm v8-A (or higher) devices is not supported by the C-SPY **Performance Monitoring** window.
- To use **Auto-Update** on a Cortex-A or Cortex-R device, IAR Embedded Workbench also requires a probe with a driver that can connect to the PMU through a debug access point (DAP)—currently only available with the I-jet family of probes—and the target must have memory-mapped registers. If these requirements are not met, the values of the event counters will only be updated when the application execution is stopped. For a Cortex-M device equipped with a PMU, DAP is not required for enabling Auto-Update, but currently only the I-jet family of probes are officially supported by C-SPY.



When the **Performance Monitoring** window is open, the PMU registers in the **Registers** window (if visible) cannot be modified and their values cannot be reliably synced with the values in the **Performance Monitoring** window.



For Armv7 and Armv8 devices, the PMU counters use 32-bit registers (for Armv8.1M devices, the PMU counters only use 16-bit registers), but the values in the **Performance Monitoring** window will show 64-bit data, taking into account counter overflows.

Event types

The PMU is designed to count different types of events. There are two categories of events—*common events* and *implementation-defined* events.

Common events

If implemented, a common event must comply with the definitions given by the Arm Architecture specification. The **Performance Monitoring** window provides a standard list of these events. If a common event is not implemented in the target device, the counter will not increment. Therefore, you are responsible for knowing whether or not an event has been implemented in the device you are using.

Implementation-defined events

An implementation-defined event is more dependent on the target device, which gives the chip manufacturer more freedom in the design. However, Arm strongly recommends that the implementation conforms to the definitions in the Arm Architecture specification.

The implementation-defined event list in C-SPY can contain events specific to the current device. The **Performance Monitoring** window manages functions for loading and saving this list as a text file that you can create and modify. Examples of architecture specific implementation-defined event lists are provided in the directory `arm\src\PMU`. This list is also automatically saved in the project settings file in the current IAR Embedded Workbench project folder when C-SPY exits.

Detecting counter overflow

Counter overflow occurs when the highest bit (bit 31 on 32-bit registers) in a PMU counter changes from `1` to `0`. This change can be seen in the `PMOVSR` register in **32-bit mode**, and in the `PMOVSR_EL0` register in **64-bit mode** (which is only available for Armv8 devices). For Armv8.1M devices, the register is named `PMU_OVSR`.

If **Auto-Update** is enabled, the overflows will be incremented when detected, and shown in the **Overflow** column in the **Performance Monitoring** window.

If **Auto-Update** is disabled, C-SPY will not be able to keep track of the overflows, which makes the values of the counters in the **Value** column unreliable. Whenever the execution is paused and an overflow is detected in a counter, there is no way for C-SPY to know if there has been more than one overflow during execution. In general, the values in the **Value** column are unreliable whenever the corresponding counters show a value less than zero (0) in the **Overflow** column.

Occasionally, a value in the **Value** column might show an asterisk (*) before the actual value and a tooltip with the word **Overflow** when the mouse pointer hovers over the affected counter value. This indicates that the overflow bit for that counter is set, but C-SPY has not yet updated the value and incremented the overflow count in the **Overflow** column.

SETTING UP PERFORMANCE MONITORING

To monitor event counters or CPU cycles through the PMU in the **Performance Monitoring** window, you must make settings in the window and select the events that you would like to monitor.

To set up performance monitoring:

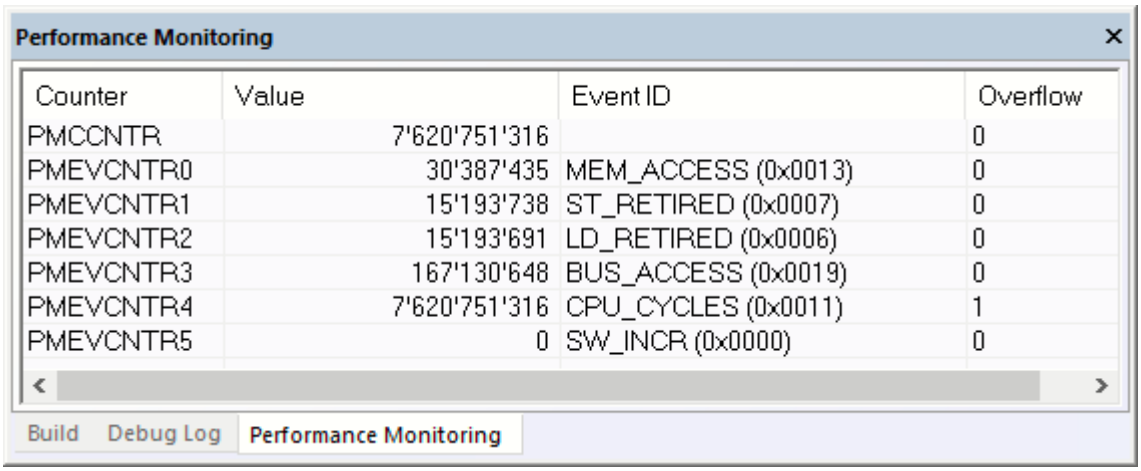
1. When the debug probe is connected, choose *Driver*>**Performance Monitoring**.
2. In the **Performance Monitoring** window, right-click and choose **Enable Performance Monitoring** from the context menu.
3. For each counter, use the context menu options to select a common or implementation-defined event that you want to monitor.

When you run the application, the event counters or CPU cycles through the PMU are displayed in the **Performance Monitoring** window.

REFERENCE INFORMATION ON PERFORMANCE MONITORING

Performance Monitoring window

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



Use this window to monitor counting events or CPU clock cycles through the PMU.

The names of the counters in this window can be used as pseudo-variables in other windows, such as a Watch-type window.

Requirements

A debug probe with support for performance monitoring and one of these alternatives:

- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver

Display area

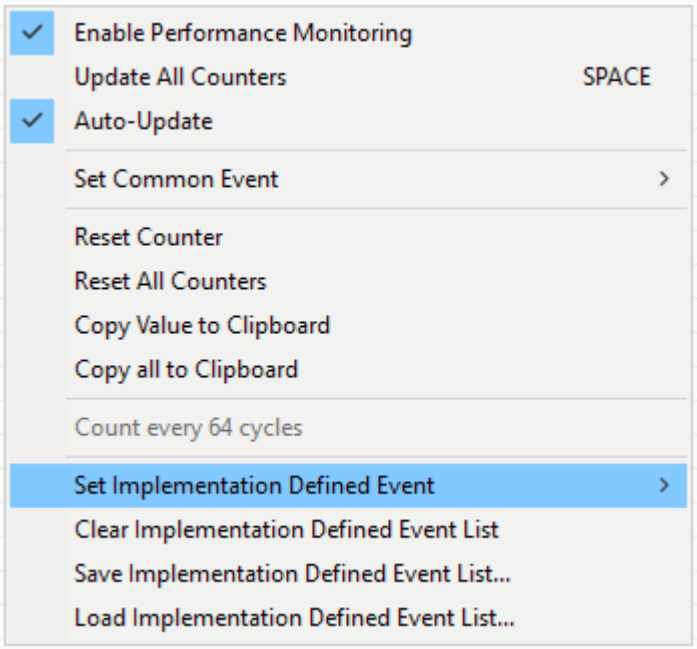
This area contains a description of the most recent fault exception that was encountered during application execution.

This area contains these columns:

Counter	The names of the counters.
Value	The resulting counts.
Event ID	The assigned event IDs for the counter.
Overflow	The number of detected overflows.

Context menu

This context menu is available for setting up and managing events and counter results:



Enable Performing Monitoring Enables performance monitoring.

Update All Counters If the PMU can be accessed at runtime (that is, when the PMU registers are memory-mapped and are accessible at runtime), this menu command refreshes the values of all counters, including `PMCCNTR`, otherwise it will be unavailable.

Auto-Update This option is only available when the PMU registers can be accessed at runtime. If enabled, the **Performance Monitoring** window automatically refreshes at regular intervals. The time between each refresh is controlled by the **LiveWatch** setting on the **Tools>Options>Debugger** page.

During execution, C-SPY also accumulates overflows and adds them to the total value of each respective counter.

Note that if the **Auto-Update** refresh rate is set too low, a PMU register might overflow more than once during execution, which will make the total counts unreliable. Make sure that the updates are made at least once for the smallest register, assuming that it increments with the same frequency as the cycle counter.

Set Common Event Opens the list of common events. The selected event will be assigned to the currently selected counter.

Note that this menu item does not work for `PMCCNTR`.

Reset Counter Resets the currently selected counter. This operation also resets the Overflow for that counter in the `PMOVSR` register.

Reset All Counters Resets all counters, including `PMCCNTR`. This operation also resets all Overflow bits in the `PMOVSR` register.

Copy Value to Clipboard Copies the value of the currently selected counter excluding the overflow asterisk (*) marker.

Copy all to Clipboard Copies the entire contents of the window. The operation creates a table where each column is separated by a Tab character, and each line contains a counter.

Count every 64 cycles When selected, sets the `PMCR.D` bit, which makes the `PMCCNTR` only count every 64th clock cycle. This is useful to make the counting last longer before the overflow bit is set. C-SPY automatically multiplies the number of counts by 64. Note that this decreases the accuracy of the cycle counting.

Set Implementation Defined Event Opens a menu with implementation-defined events. The selected event will be assigned to the currently selected counter.

Note that this menu item does not work for `PMCCNTR`.

Clear Implementation Defined Event List Clears the list of implementation-defined events.

Save Implementation Defined Event List Opens a standard **File Save** dialog box from which you can choose a filename to save the current implementation-defined event list. The saved file is a text file where each line contains the ID, name, and description of an event, separated by spaces or tabs. The ID is stored as a hexadecimal number beginning with `0x`.

Load Implementation Defined Event List Opens a standard **File Open** dialog box from which you can load a list of implementation-defined events, where each line should contain the ID, name, and description of an event, separated by spaces or tabs.

The format is defined as this:

- The first element must be a decimal or hexadecimal number (where hexadecimal numbers must start with `0x`)
- The second element must be a name containing letters, numbers, or underscores
- The last element can contain any text, but C-SPY will trim off the starting and ending spaces when the file is loaded
- If a line does not follow this syntax, it will be silently ignored.

For example, the first line in a file containing the Armv8 implementation-defined event list should look like this:

```
0x0040  L1D_CACHE_RD  Attribute Level 1 data cache access, read
```

Note that there are no restrictions to the type of spacing used between the elements in a line. However, a Tab character is usually preferred for editing purposes.

Power debugging

Contents

Introduction to power debugging 243
Reasons for using power debugging 243
Briefly about power debugging 243
Requirements and restrictions for power debugging 244
Optimizing your source code for power consumption 244
Waiting for device status 245
Software delays 245
DMA versus polled I/O 245
Low-power mode diagnostics 245
CPU frequency 245
Detecting mistakenly unattended peripherals 246
Peripheral units in an event-driven system 246
Finding conflicting hardware setups 247
Analog interference 247
Debugging in the power domain 247
Displaying a power profile and analyzing the result 248
Detecting unexpected power usage during application execution 248
Changing the graph resolution 249
Reference information on power debugging 249
Power Log Setup window 249
Power Log window 251
Timeline window—Power graph 254
State Log window 255
State Log Summary window 256
Timeline window—State Log graph 259

INTRODUCTION TO 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 244*.

Briefly about power debugging

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

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

Measuring power consumption

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

The TI MSP-FET and TI XDS 110 debug probes use EnergyTrace™ Technology support to measure the power supplied to a target microcontroller. A software-controlled DC–DC converter generates the target power supply. The time density of the DC–DC converter charge pulses equals the power consumption of the target microcontroller. A built-in on-the-fly calibration circuit defines the energy equivalent of a single DC–DC charge pulse.

Power debugging using C-SPY

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

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

Requirements and restrictions for power debugging

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

- An I-jet or I-jet Trace in-circuit debugging probe. Note that power debugging is not possible when using I-jet Trace with ETM.

  For more accurate power debugging, you can connect the I-scope probe between the I-jet/I-jet Trace probe and the target board. I-scope adds detailed current and voltage measurement capability.
- A J-Link Ultra debug probe and a Cortex-M device with SWO.
- A TI MSP-FET debug probe, featuring the EnergyTrace™ technology provided by Texas Instruments, and a TI MSP-FET device. The probe outputs voltage, current, and energy information.
- An ST STLINK-V3PWR debug probe and a Cortex-M device with SWO. The target board should be powered via the probe.
- A TI XDS110 debug probe, featuring the EnergyTrace™ technology provided by Texas Instruments, and a Texas Instruments device. The probe outputs voltage, current, and energy information.

OPTIMIZING YOUR SOURCE CODE FOR POWER CONSUMPTION

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

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 * U2 * k
```

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

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

Detecting mistakenly unattended peripherals

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

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

Peripheral units in an event-driven system

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

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



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

The functionality of the system could be excellent and it can be optimized in terms of speed and code size. But in the power domain, more optimizations can be made. The shadowed area represents the energy that

could have been saved if the peripheral devices that are not used between t_2 and t_3 had been turned off, or if the priorities of the two tasks had been changed.

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

Finding conflicting hardware setups

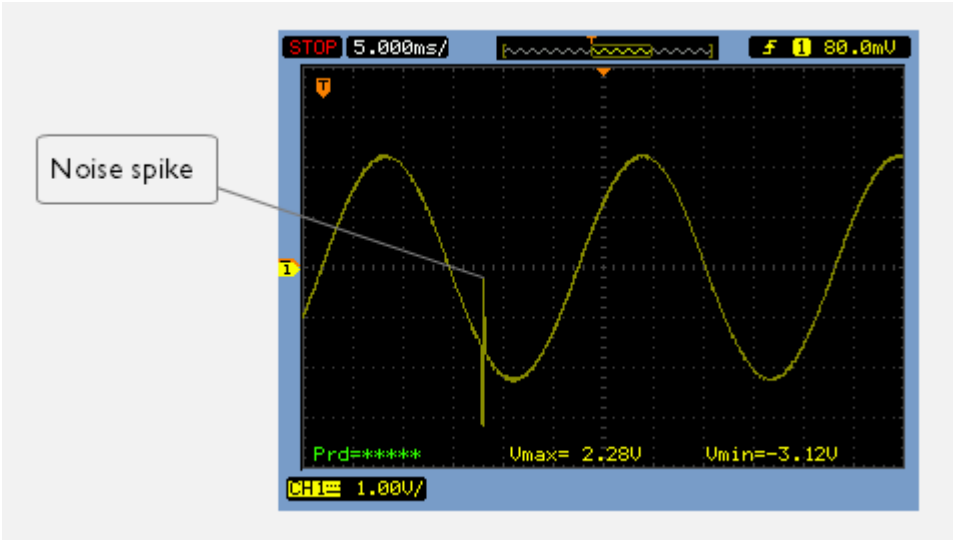
To avoid floating inputs, it is a common design practice to connect unused MCU I/O pins to ground. If your source code by mistake configures one of the grounded I/O pins as a logical `1` output, a high current might be drained on that pin. This high unexpected current is easily observed by reading the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

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

Analog interference

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

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



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

DEBUGGING IN THE POWER DOMAIN

See also:

- *Timeline window—Power graph, page 254*

- *Selecting a time interval for profiling information, page 228*

Displaying a power profile and analyzing the result

To view the power profile:

1. Choose ***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.

   This step requires a device with SWD SWO (Serial Wire Output) capability.
2. Start the debugger.
3. Choose ***Driver*>Power Log Setup**. In the **ID** column, make sure to select the alternatives for which you want to enable power logging.
4. Choose ***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 ***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 111*.

   This step requires a device with SWD SWO (Serial Wire Output) capability.
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 221*.
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 device with SWD SWO (Serial Wire Output) capability, of the data and interrupt logs if you enabled these graphs. For information about how to navigate on the graph, see *Navigating in the graphs, page 197*.
10. To analyze power consumption (requires a device with SWD SWO capability):
    - 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 244*.
    - 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 246*.
    - 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 228*.

Detecting unexpected power usage during application execution

To detect unexpected power consumption:

1. Choose ***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 the correct data transfer rate to the debug probe.

   This step requires a Cortex-M device with SWO.

2. Choose *Driver*>**Power Log Setup** to open the **Power Log Setup** window.
3. In the **Power Log Setup** window, specify a threshold value and the appropriate action, for example **Log All and Halt CPU Above Threshold**.
4. Choose *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.

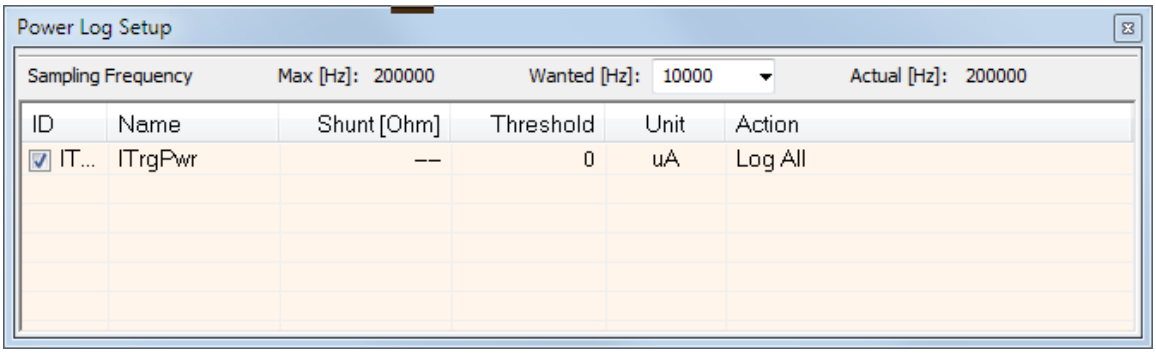
REFERENCE INFORMATION ON POWER DEBUGGING

See also:

- *Trace window, page 170*
- *The application timeline, page 195*
- *Viewing Range dialog box, page 221*
- *Function Profiler window, page 229*

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.

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.

See also *Debugging in the power domain, page 247*.

Requirements

A debug probe with support for power debugging and one of these alternatives:

- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver
- The C-SPY TI MSP-FET driver
- The C-SPY TI XDS driver

Sampling Frequency toolbar

Use this toolbar to override the default sampling frequency, and to inspect the maximum and the actual sampling frequency.

Max [Hz] This is the maximum supported sampling frequency in Hertz.

Wanted [Hz] Use this option to override the default sampling frequency. The frequency you enter will be rounded to the nearest possible value, and the **Actual [Hz]** value will be updated accordingly.

Actual [Hz] This is the actual sampling frequency being used.

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.

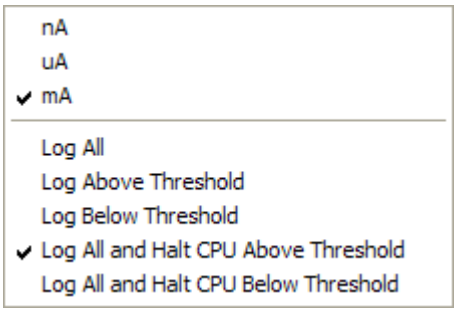
Unit Displays the selected unit for power. You can choose a unit from the context menu.

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

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

Context menu

This context menu is available:

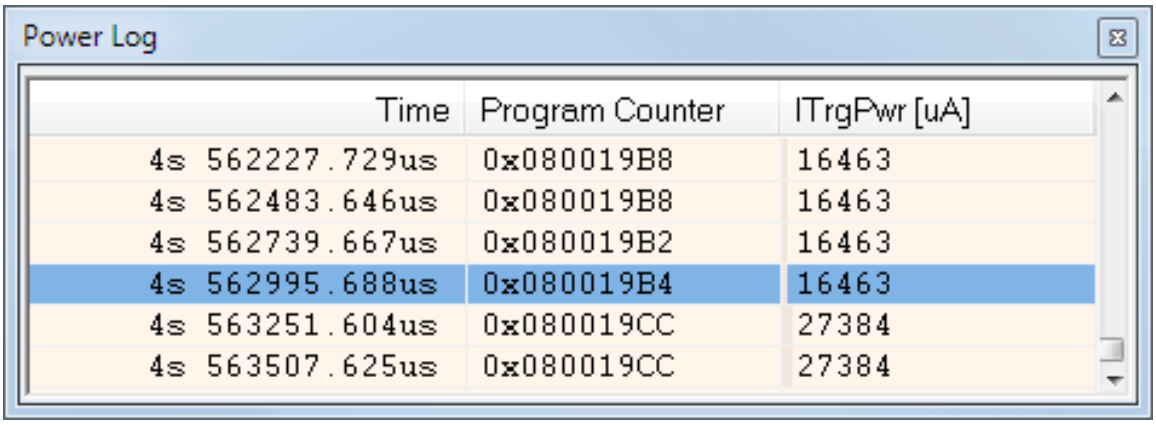


These commands are available:

nA, uA, mA	Selects the unit for the power display. These alternatives are available for channels that measure current.
uV, mV, V	Selects the unit for power display. These alternatives are available for channels that measure voltage.
uWs, mWs, Ws	Selects the unit for power display. These alternatives are available for channels that measure energy.
Log All	Logs all values.
Log Above Threshold	Logs all values above the threshold.
Log Below Threshold	Logs all values below the threshold.
Log All and Halt CPU Above Threshold	Logs all values. If a logged value exceeds the threshold, execution is stopped. This might take a few execution cycles.
Log All and Halt CPU Below Threshold	Logs all values. If a logged value goes below the threshold, execution is stopped. This might take a few execution cycles.

Power Log window

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



This window displays collected power values.

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

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

See also *Debugging in the power domain, page 247*.

Requirements

A debug probe with support for power debugging and one of these alternatives:

- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver
- The C-SPY TI MSP-FET driver
- The C-SPY TI XDS 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.

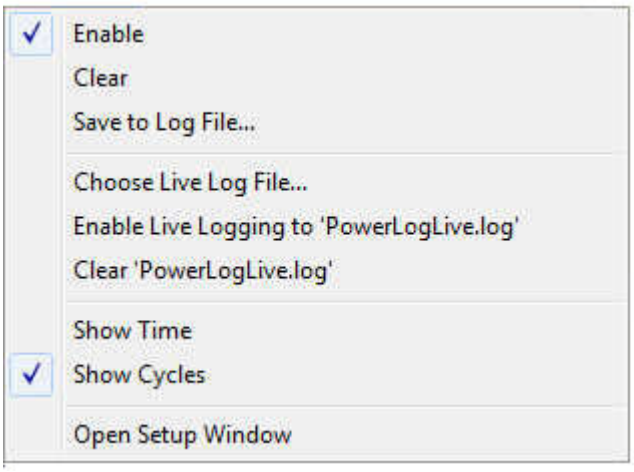
Cycles
The number of cycles from the application reset until the event. This information is cleared at reset.
If a cycle is displayed in italics, the target system could not collect a correct time, but instead had to approximate it.
This column is available when you have selected **Show Cycles** from 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.
Note that the **Program Counter** column is only available if your C-SPY debugger driver and device support it.

***Name* [*unit*]**
The power measurement value expressed in the unit you specified in the **Power Log 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	Deletes the log information. Note that this will also happen when you reset the debugger, or if you change the execution frequency in the **SWO Configuration** dialog box.
Save to File	Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Choose Live Log File	Displays a standard file selection dialog box where you can choose a destination file for the logged power values. The power values are continuously saved to that file during execution. The content of the live log file is never automatically cleared, the logged values are simply added at the end of the file.
Enable Live Logging to	Toggles live logging on or off. The logs are saved in the specified file.
Clear *log file*	Clears the content of the live log file.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.
Open Setup Window	Opens the **Power Log Setup** window.

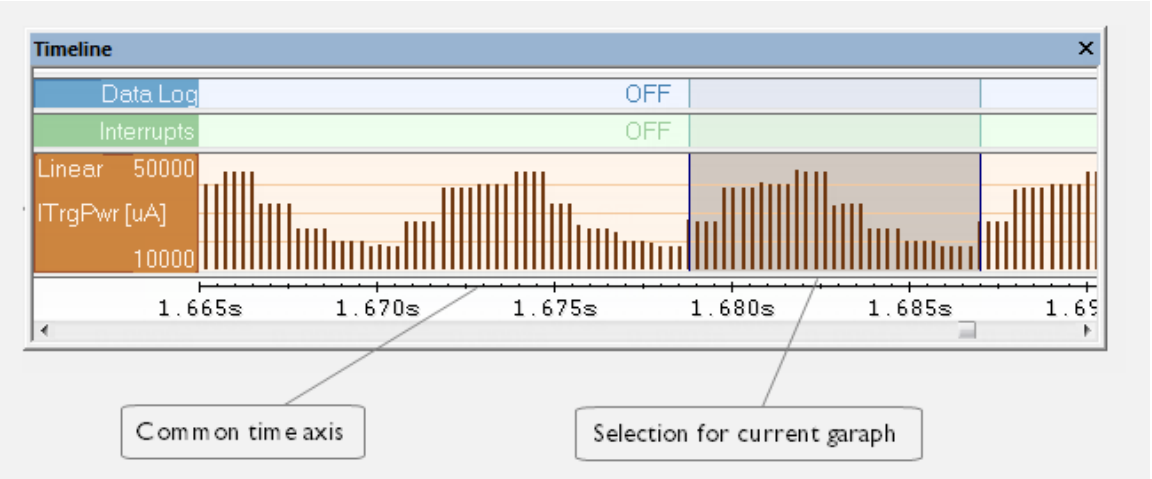
The format of the log file

The 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.

Timeline window—Power graph

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



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

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

For more information about the **Timeline** window, how to display a graph, and the other supported graphs, see *The application timeline, page 195*.

See also *Requirements and restrictions for power debugging, page 244*.

Requirements

A debug probe with support for power debugging and one of these alternatives:

- The C-SPY I-jet driver
- The C-SPY J-Link/J-Trace driver
- The C-SPY ST-LINK driver
- The C-SPY TI MSP-FET driver
- The C-SPY TI XDS driver

Display area

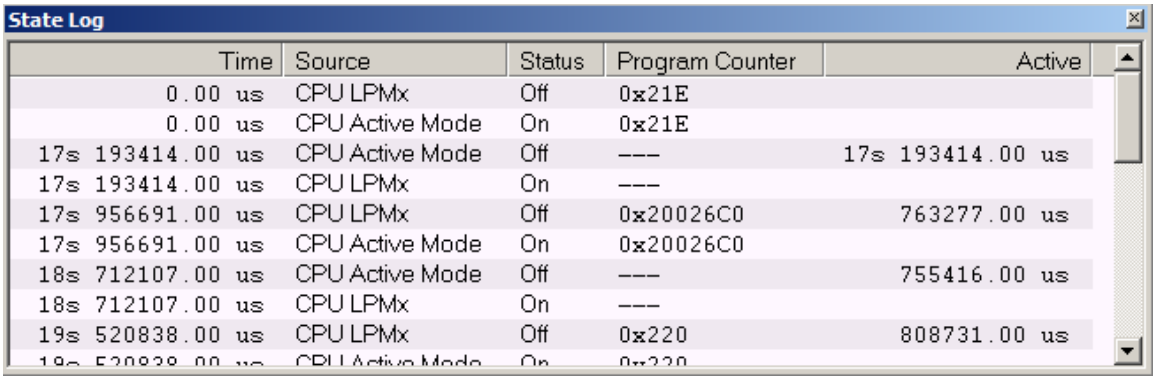
Where:

- The label area at the left end of the graph displays the name of the measurement channel.
- The Voltage and the Current graphs show power measurement samples generated by the debug probe or associated hardware. The Energy graph shows accumulated energy since the last time the CPU was stopped.
- 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. Note that if you change the setting to a very coarse resolution ("zooming out"), the same point in the graph can come to represent two or more values. In that case two values will be shown for that point on the X axis. The darker one is the lowest value for that point and the lighter one is the highest value for that point.
- 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.

State Log window

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



This window logs activity—state changes—for peripheral units and clocks, as well as for CPU modes.

The information is useful for tracing the activity on the target system. When the **State Log** window is open, it is updated continuously at runtime.

The number of saved logs is limited. When this limit is exceeded, the entries at the beginning of the buffer are erased.

See also *Displaying a power profile and analyzing the result, page 248* and *Timeline window—State Log graph, page 259*.

Requirements

One of these alternatives:

- The C-SPY MSP-FET driver and a TI MSP-FET debug probe with EnergyTrace+™ Technology provided by Texas Instruments.
- The C-SPY TI XDS driver and a TI XDS110 debug probe with EnergyTrace+™ Technology provided by Texas Instruments, and one of the MSP432, CC13xx, or CC26xx devices.

Display area

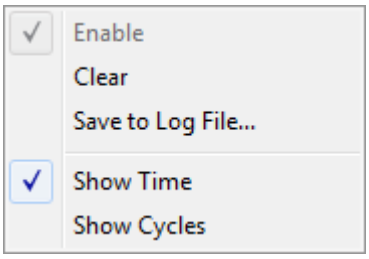
This area contains these columns:

Time	The time for the state change, based on the sampling frequency. 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.
Source	The name of the peripheral unit, clock, or CPU mode.
Status	The status at the given time.
Program Counter*	The address of the program counter when the status changed, or shows **idle** if the log was taken during CPU idle mode, or shows **---** for an unknown `PC` value.
Active	The active time calculated using the on and off time for the source. If it is written in italics, it is based on at least one approximative time.

* 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).

Context menu

This context menu is available:

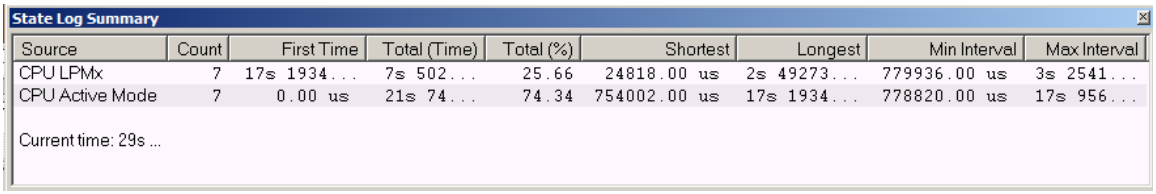


These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Save to 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` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

State Log Summary window

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



This window displays a summary of logged activity—state changes—for peripheral units and clocks, as well as for CPU modes.

Click a column to sort it according to the values. Click again to reverse the sort order.

At the bottom of the display area, the current time or cycles is displayed—the number of cycles or the execution time since the start of execution.

See also *Displaying a power profile and analyzing the result, page 248* and *Timeline window—State Log graph, page 259*.

Requirements

One of these alternatives:

- The C-SPY MSP-FET driver and a TI MSP-FET debug probe with EnergyTrace+™ Technology provided by Texas Instruments.
- The C-SPY TI XDS driver and a TI XDS110 debug probe with EnergyTrace+™ Technology provided by Texas Instruments, and one of the MSP432, CC13xx, or CC26xx devices.

Display area

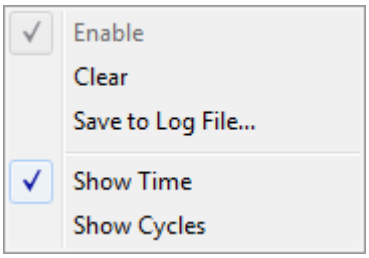
Each row in this area displays statistics about the specific measurement source based on the log information in these columns; and summary information is listed at the bottom of the display area:

Source	The name of the peripheral unit, clock, or CPU mode.
Count	The number of times the source was activated.
First time	The first time the source was activated.
Total (Time)**	The accumulated time the source has been active.
Total (%)	The accumulated time in percent that the source has been active.
Shortest	The shortest time spent with this source active.
Longest	The longest time spent with this source active.
Min interval	The shortest time between two activations of this source.
Max interval	The longest time between two activations of this source.
Approximative time count	The information displayed depends on the C-SPY driver you are using. For some C-SPY drivers, this information is not displayed or the value is always zero. In this case, all logs have an exact time stamp. For other C-SPY drivers, a non-zero value is displayed. The value represents the amount of logs with an approximative time stamp. This might happen if the bandwidth in the communication channel is too low compared to the amount of data packets generated by the CPU or if the CPU generated packets with an approximative time stamp.
Overflow count	The information displayed depends on the C-SPY driver you are using. For some C-SPY drivers, this information is not displayed or the value is always zero. For other C-SPY drivers, the number represents the amount of overflows in the communication channel which can cause logs to be lost. If this happens, it indicates that logs might be incomplete. To solve this, make sure not to use all C-SPY log features simultaneously or check used bandwidth for the communication channel.
Current time/Current cycles	The information displayed depends on the C-SPY driver you are using. For some C-SPY drivers, the value is always zero or not visible at all. For other C-SPY drivers, the number represents the current time or cycles—the number of cycles or the execution time since the start of execution.

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

Context menu

This context menu is available:

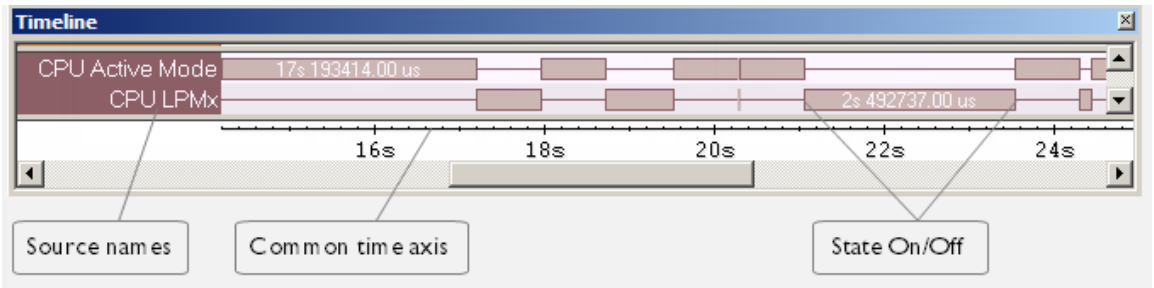


These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Save to 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` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Timeline window—State Log graph

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



The **State Log** graph displays a graphical view of logged activity—state changes—for peripheral units and clocks, as well as CPU modes 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 *The application timeline, page 195*.

See also *Requirements and restrictions for power debugging, page 244*.

Requirements

One of the following combinations:

- The C-SPY MSP-FET driver and a TI MSP-FET debug probe with EnergyTrace+™ Technology provided by Texas Instruments.
- The C-SPY TI XDS driver and a TI XDS110 debug probe with EnergyTrace+™ Technology provided by Texas Instruments, and one of the MSP432, CC13xx, or CC26xx devices.

Display area

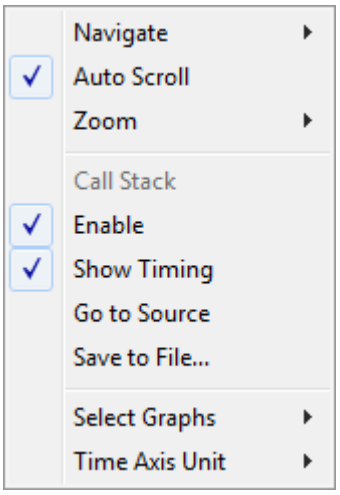
Where:

- The label area at the left end of the graph displays the name of the sources of the status information.
- The graph itself shows the state of the peripheral units, clocks, and CPU modes generated by the debug probe or associated hardware. The white figure indicates the time spent in the state. This graph is a graphical representation of the information in the **State Log** window, see *State Log window, page 255*.

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

Context menu

This context menu is available:



The context menu contains some commands that are common to all graphs in the **Timeline** window and some commands that are specific to each graph.

These commands are available:

Navigate — Commands for navigating the graph(s). Choose between:

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

Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll — Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.

Zoom — Commands for zooming the window, in other words, changing the time scale. Choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +

Zoom Out zooms out on the time scale. Shortcut key: –

10ns, **100ns**, **1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

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

10m, **1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Power Log — A heading that shows that the Power Log-specific commands below are available.

Enable — Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.

Viewing Range	Displays a dialog box, see *Viewing Range dialog box, page 221*.
Size	Determines the vertical size of the graph—choose between **Small**, **Medium**, and **Large**.
Style	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	Shows the numerical value of the variable, in addition to the graph.
Go To Source	Displays the corresponding source code in an editor window, if applicable.
Open Setup Window	Opens the **Power Log Setup** window.
Select Graphs	Selects which graphs to be displayed in the **Timeline** window.
Time Axis Unit	Selects the unit used in the time axis—choose between **Seconds** and **Cycles**. If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.
Profile Selection	Enables profiling time intervals in the **Function Profiler** window. Note that this command is only available if the C-SPY driver supports PC Sampling. See *Selecting a time interval for profiling information, page 228*.

C-RUN runtime error checking

Contents

Introduction to runtime error checking 264
Runtime error checking 264
Runtime error checking using C-RUN 264
The checked heap provided by the library 265
Using C-RUN in the IAR Embedded Workbench IDE 265
Using C-RUN in non-interactive mode 265
Requirements for runtime error checking 266
Using C-RUN 266
Getting started using C-RUN runtime error checking 266
Creating rules for messages 267
Detecting various runtime errors 267
Detecting implicit or explicit integer conversion 267
Detecting signed or unsigned overflow 269
Detecting bit loss or undefined behavior when shifting 270
Detecting division by zero 271
Detecting unhandled cases in switch statements 272
Detecting accesses outside the bounds of arrays and other objects 273
Detecting heap usage error 277
Detecting heap memory leaks 279
Detecting heap integrity violations 280
Reference information on runtime error checking 283
C-RUN Runtime Checking options 283
C-RUN Messages window 285
C-RUN Messages Rules window 287
Compiler and linker reference for C-RUN 288
--bounds_table_size 288
--debug_heap 289
--generate_entries_without_bounds 289
--ignore_uninstrumented_pointers 289
--ignore_uninstrumented_pointers 290
--runtime_checking 290
#pragma default_no_bounds 291
#pragma define_with_bounds 291
#pragma define_without_bounds 291
#pragma disable_check 292
#pragma generate_entry_without_bounds 292
#pragma no_arith_checks 292
#pragma no_bounds 293
__as_get_base 293
__as_get_bound 293
__as_make_bounds 294
cspybat options for C-RUN 294
--rtc_enable 294
--rtc_output 294
--rtc_raw_to_txt 295
--rtc_rules 295

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

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`.



C-RUN depends on the Arm semi-hosting interface (the library function `__iar_ReportCheckFailed` will communicate with C-SPY via the semihosting interface). It is only in non-interactive mode that you can use another low-level I/O interface. See *Using C-RUN in non-interactive mode, page 265*.

For information about how to detect the errors, see *Detecting various runtime errors, page 267*.

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 285*.
- The **C-RUN Message Rules** window, which lists all rules. See *C-RUN Messages Rules window, page 287*. 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`.

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, because it only contains the raw data. You can use `cspybat` with the simulator driver to transform the raw text into something very similar to normal C-RUN messages. For more information, see *--rtc_raw_to_txt, page 295*.

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 294*.

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

Getting started using C-RUN runtime error checking

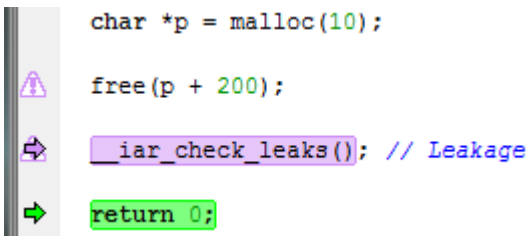
Typical use of C-RUN involves these steps:

- 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.

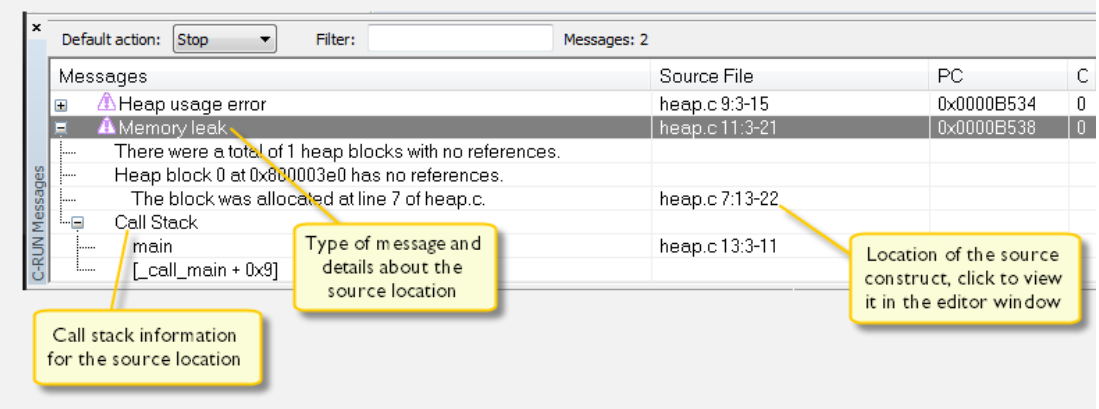
More in detail, to perform runtime error checking and detect possible runtime errors, follow this example of a typical process:

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:



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 267*.

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.



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

DETECTING VARIOUS RUNTIME ERRORS

Detecting implicit or explicit integer conversion

Description

Checks that an integer conversion (implicit or explicit) or a write access to a bitfield does not change the value.

Why perform the check

Because C allows converting larger types to smaller integer types, some conversions can unintentionally remove significant bits of the value. The check can be limited to implicit integer conversions, which is useful when the loss of data caused by explicit conversion is considered intentional.

How to use it

Compiler option: `--runtime_checking integer_conversion|implicit_integer_conversion`

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

The check can be applied to one or more modules.

The check can be avoided by inserting an explicit mask:

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

How it works

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

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

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

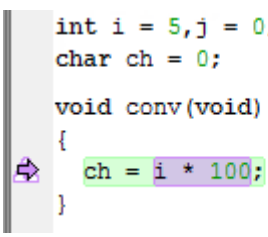
```
signed char a = 127;
void f(void)
{
  ++a;    /* Conversion check error (128 -> -128) */
  a -= 1; /* Conversion check error (-129 -> 127) */
}
```

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

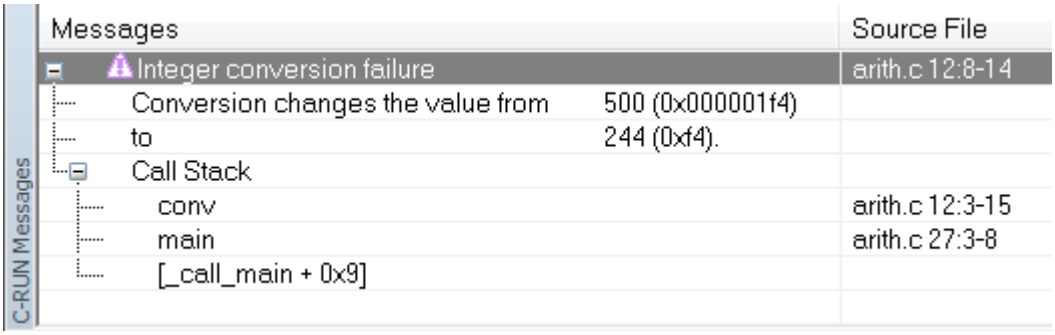
Example

Follow the procedure described in *Getting started using C-RUN runtime error checking, page 266*, but use the **Integer conversion** option.

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



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



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 *Detecting bit loss or undefined behavior when shifting, page 270*.

Why perform the check

Because the behavior of signed overflow is undefined, and because `unsigned` overflow results in a truncation that can sometimes be undesirable. Although the shift operation is not checked, shift counts are checked because if a shift count is negative or greater than or equal to the width of the promoted left operand, the behavior of the shift operation is undefined.

How to use it

Compiler option: `--runtime_checking signed_overflow|unsiged_overflow`

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

The check can be applied to one or more modules.

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

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

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.

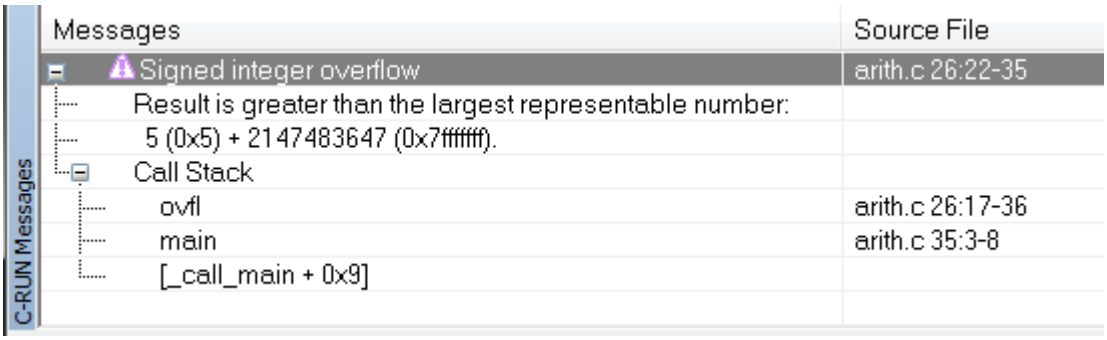
Example

Follow the procedure described in *Getting started using C-RUN runtime error checking, page 266*, 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 ingeger overflow`, `Unsigned integer overflow`, or `Shift count overflow`. This is an example of the message information that will be listed:



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

Because the behavior of signed overflow is undefined, and because unsigned overflow results in a truncation that can sometimes be undesirable.

Overflow occurs in a left shift operation `E1<<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.

How to use it

Compiler option: `--runtime_checking signed_shift|unsigned_shift`

In the IDE: **Project>Options>Runtime Checking>Integer shift overflow**

The check can be applied to one or more modules.

The check can be avoided by masking before shift:

```
/* Cannot overflow */
int f(int x) { return (x & 0x00007FFF) << 16; }
```

How it works

The compiler inserts code to perform the check for each shift operation, unless the compiler determines that the check cannot fail.

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

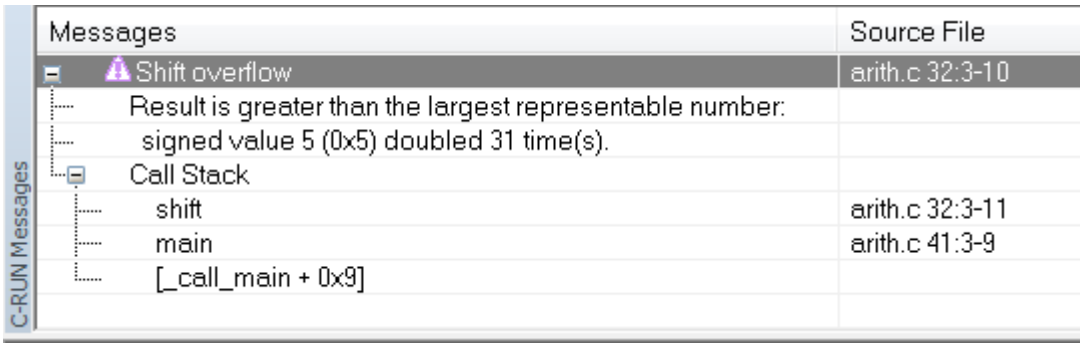
Example

Follow the procedure described in *Getting started using C-RUN runtime error checking, page 266*, but use the **Integer shift overflow** option.

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

```
void shift(void)
{
  i <<= 31;
}
```

C-RUN will report either `Shift overflow` or `Shift count overflow`. This is an example of the message information that will be listed:



Detecting division by zero

Description

Checks for division by zero and modulo by zero. Floating-point operations are checked for division by exactly (positive) zero.

Why perform the check

Because the behavior of integer division by zero is undefined, and because floating-point division by exactly zero usually indicates a problem.

How to use it

Compiler option: `--runtime_checking div_by_zero`

In the IDE: **Project>Options>Runtime Checking>Division by zero**

The check can be applied to one or more modules.

How it works

The compiler inserts code to perform the check at each division and modulo operation, unless the compiler determines that the check cannot fail.

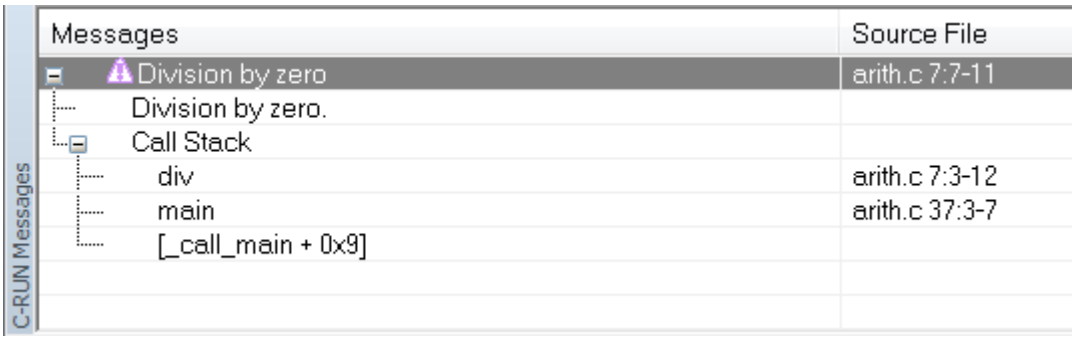
Example

Follow the procedure described in *Getting started using C-RUN runtime error checking, page 266*, but use the **Division by zero** option.

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

```
void div(void)
{
  j = i / j;
}
```

C-RUN will report `Division by zero`. This is an example of the message information that will be listed:



Detecting unhandled cases in switch statements

Description

Checks for a missing `case` label in a `switch` statement that does not have a default label.

Why perform the check

The check is useful, for example, to detect when an `enum` type has been augmented with a new value that is not yet handled in a `switch` 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.

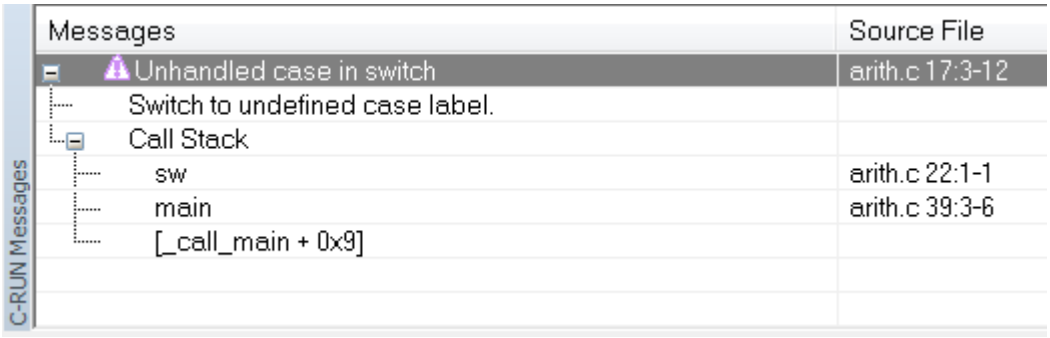
Example

Follow the procedure described in *Getting started using C-RUN runtime error checking, page 266*, 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:



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.
- All floating-point library functions will use the softfp interface.

For each access through a pointer expression, the calculated address and the calculated address plus the access size is checked against the bounds. If any of the two addresses are outside of the bounds, a C-RUN message is generated.

Functions that receive pointers in any parameters, or that return a pointer value, can exist in two variants, one with the bounds, and one without the bounds.

Resource usage

The bounds checking overhead can cause the application to no longer fit in the available ROM or RAM. There are some ways you can try to deal with this:

- Provided that your application does not use too many indirectly accessed pointers, you can shrink the global bounds table to reduce the amount of RAM used for it. See *--bounds_table_size, page 288* (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 does not. See the next section for more information.

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 acesses 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_unistrumented_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 * gptr;
int main(void)
{
  /* Give gptr bounds with size N. */
  gptr = __as_make_bounds(gptr, N);
  ...
}
```

- *RTOS tasks*

  The function that implements a task might get called with a parameter that is a pointer. If the RTOS itself is not tracking pointer bounds, you must use `#pragma define_without_bounds` and `__as_make_bounds` to get the correct bounds information.

  For example:

```
#pragma define_without_bounds
void task1(struct Arg * p)
{
  /* p points to a single Arg struct */
  p = __as_make_bounds(p, 1);
  ...
}
```

Some limitations:

- *Function pointers*

  Sharing a function pointer between code that tracks bounds and code that does not can be problematic.

  There is no difference in type between functions that track bounds, and functions that do not. Functions of both kinds can be assigned to function pointers, or passed to functions that take function pointer parameters. However, if a function whose signature includes pointers is called in a non-matching context (a function that tracks bounds from code that does not, or vice versa), things will not work reliably. In the most favorable cases, this will mean confusing bounds violations, but it can cause practically any behavior because these functions are being called with an incorrect number of arguments.

  For things to work, you must ensure that all functions whose signature includes pointers, and which are called via function pointers, are of the right kind. For the simple case of call-backs from a library that does not track bounds, it will usually suffice to use `#pragma no_bounds` on the relevant functions.

- *K&R functions*

  Do not use K&R functions. Use `--require_prototypes` and shared header files to make sure that all functions have proper prototypes. Note that in C `void f()` is a K&R function, while `f(void)` is not.

- *Pointers updated by code that does not track bounds*

  Whenever a pointer is updated by code that does not set up new bounds for the pointer, there is a potential problem. If the new pointer value does not point into the same object as the old pointer value, the bounds will be incorrect and an access via this pointer in checked code will signal an error.

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 266*, but use the **Bounds checking** option.

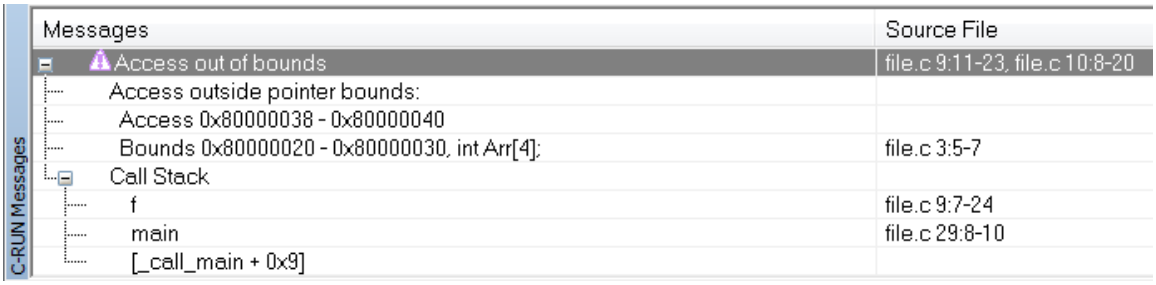
This is an example of source code that will be identified during runtime:

```
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:



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 265*.

Example

Follow the procedure described in *Getting started using C-RUN runtime error checking, page 266*, but use the **Debug heap** option.

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

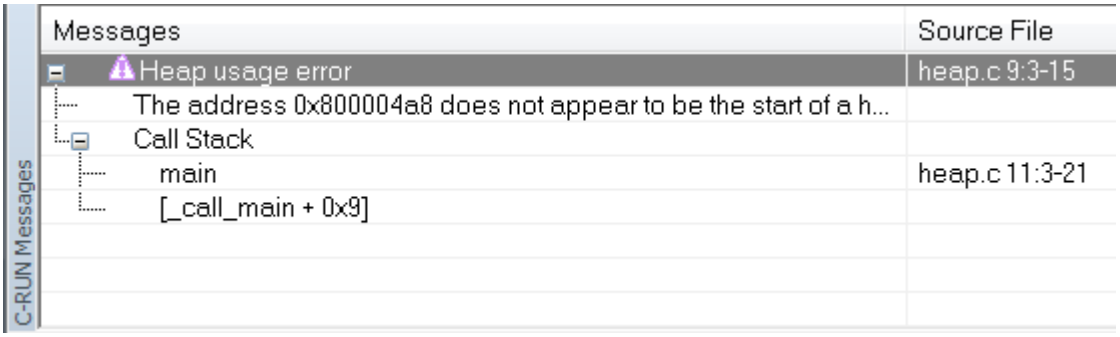
```
int main(void)
{
  char *p = malloc(10);

  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:



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.



The leak checker does not currently support multi-threaded environments. The leak checker works by scanning known RAM locations for references to heap blocks. The thread executing the leak check has information about its own stack, but not about the stack of other threads. The missing information can result in both false positives and false negatives.

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 265*.

Example

Follow the procedure described in *Getting started using C-RUN runtime error checking, page 266*, but use the **Debug heap** option.

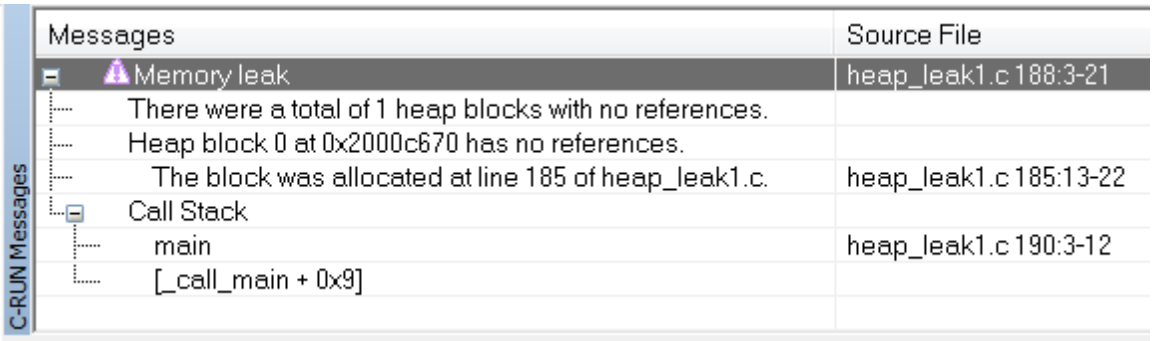
This is an example of source code that will be identified during runtime:

```
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:



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 265*.

Example

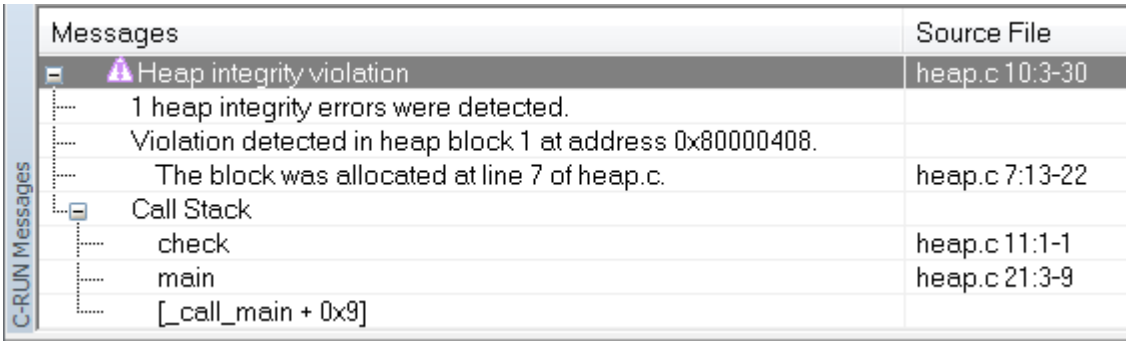
Follow the procedure described in *Getting started using C-RUN runtime error checking, page 266*, but use the **Checked heap** option.

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();
}
```

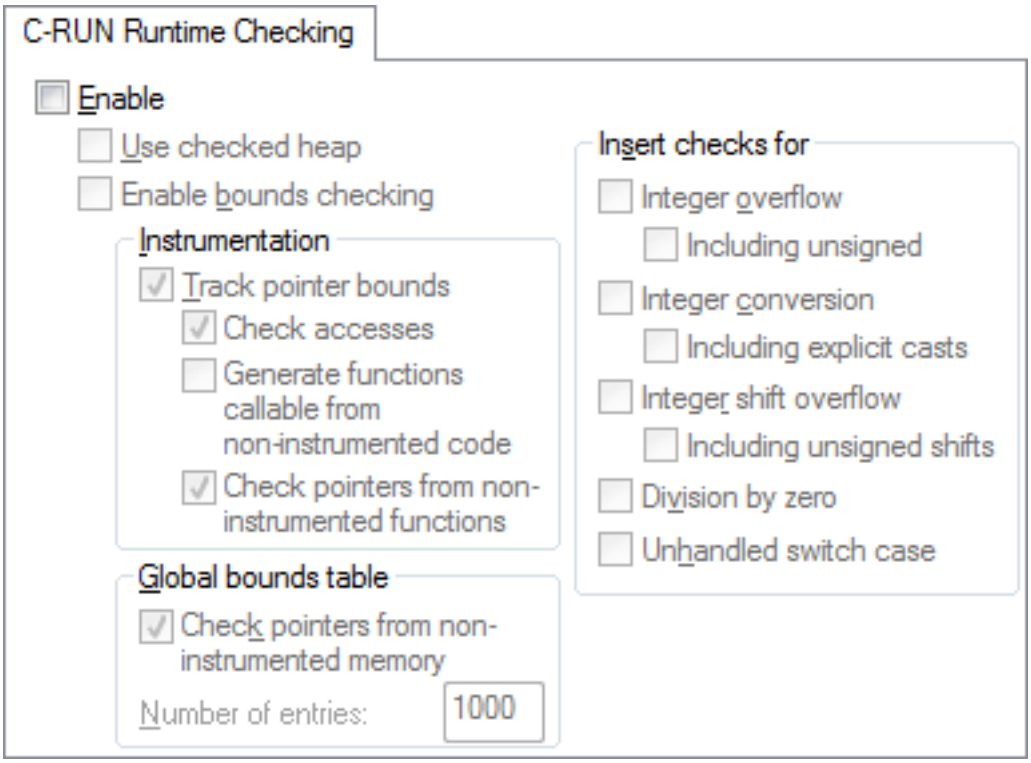
C-RUN will report `Heap integrity violation`. This is an example of the message information that will be listed:



REFERENCE INFORMATION ON RUNTIME ERROR CHECKING

C-RUN Runtime Checking options

The **C-RUN Runtime Checking** options determine which checks to perform at runtime.



See also *Using C-RUN, page 266*.

Enable

Enables runtime checking.

Use checked heap

Uses the checked heap, to detect heap usage errors.

Enable bounds checking

Checks for accesses outside the bounds of arrays and other objects. Available checks:

Track pointer bounds Makes the compiler add code that tracks pointer bounds. If you want to check pointer bounds, you should enable **Check accesses** and then decide how instrumented code should interact with non-instrumented code:

Check accesses	Inserts code for checking accesses via pointers.
Generate functions callable from non-instrumented code	When **Track pointer bounds** is enabled, any functions that return or receive types that contain pointers are modified to also return/receive pointer bounds. Use this option to generate an extra entry for each such function, which can be called from unchecked code.
Check pointers from non-instrumented functions	When **Track pointer bounds** is enabled, pointers that originate from functions that are not instrumented for bounds checking are by default given globally permissive bounds information. Use this option to identify these pointers—any accesses via such pointers will generate an error. In this way you can manually replace the globally permissive bounds information with valid counterparts, see *__as_get_base, page 293*, *__as_get_bound, page 293*, *__as_make_bounds, page 294*. 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 globally permissive bounds. Use this option to identify such pointers—any accesses via such pointers will generate an error. In this way you can manually replace the globally permissive bounds information with valid counterparts, see *__as_get_base, page 293*, *__as_get_bound, page 293*, *__as_make_bounds, page 294*. 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.
Number of entries	The bounds checking system uses a separate table to track bounds for pointers in memory. Use this option to set the number of such bounds that can be tracked simultaneously. The table will use approximately 50 bytes per pointer.

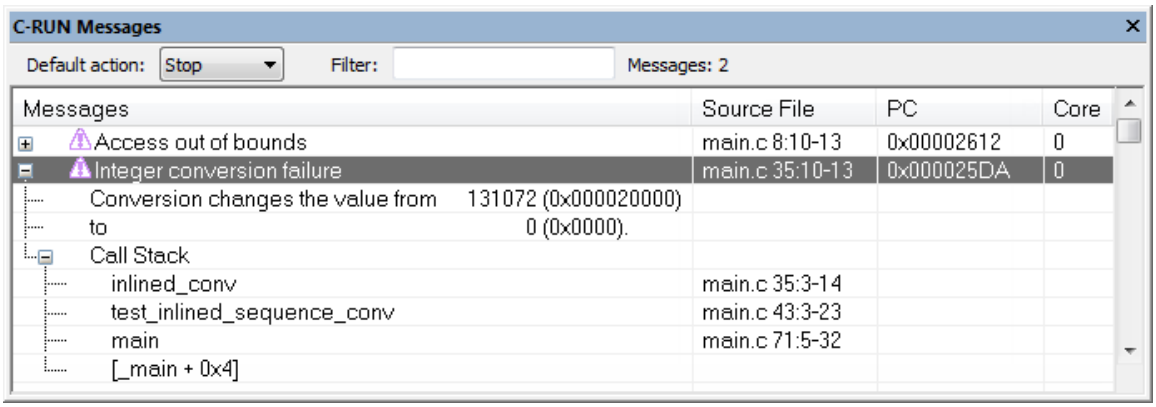
Insert checks for

Inserts checks for:

Integer overflow	Checks for signed overflow in integer operations. Use **Including unsigned** to also check for unsigned overflow in integer operations.
Integer conversion	Checks for implicit integer conversions resulting in a change of value. Use **Including explicit casts** to also check for explicit casts.
Integer shift overflow	Checks for overflow in shift operations. Use **Including unsigned shifts** to also check for unsigned overflow in shift operations.
Division by zero	Checks for division by zero.
Unhandled switch case	Checks for unhandled cases in `switch` statements

C-RUN Messages window

The **C-RUN Messages** window is available from the **View** menu.



This window displays information about runtime errors detected by a runtime check. The window groups messages that have the same source statement, the same call stack, and the same messages.

See also *Using C-RUN, page 266*.

Requirements

A license for the C-RUN product.

Toolbar

The toolbar contains:

Default action	Sets the default action for what happens if no other rule is satisfied. Choose between **Stop**, **Log**, and **Ignore**.
Filter	Filters the list of messages so that only messages that contain the text you specify will be listed. This is useful if you want to search the message text, call stack entries, or filenames.
Messages	Lists the number of C-RUN messages.

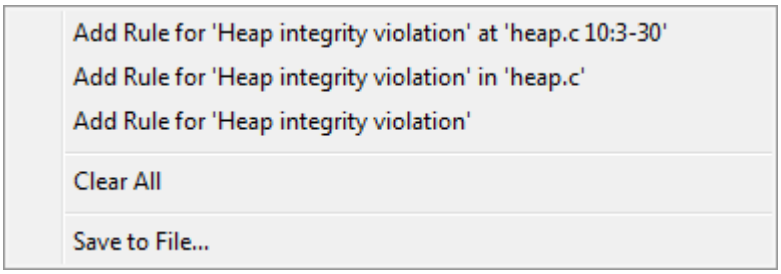
Display area

The display area shows all detected errors since the last reset. More specifically, the display area provides information in these columns:

Messages	Information about the detected runtime error. Each message consists of a headline, detailed information about the error, and call stack information for the error location. Note that ranges displayed for accesses and bounds include the start address but not the end address.
Source File	The name of the source file in which a runtime error was detected, or otherwise a relevant location, for example variable definitions.
PC	The value of `PC` when the runtime error was detected.
Core	The CPU core that executed the check, in case you have a multicore environment.

Context menu

This context menu is available:

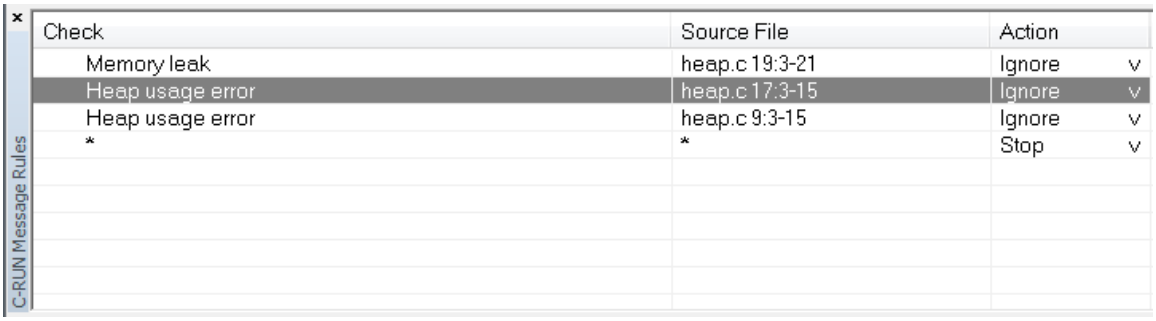


These commands are available:

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.



This window displays the rules that control how messages are reported in the **C-RUN Messages** window. When a potential error is detected, it is matched against these rules (from top to bottom) and the action taken is determined by the first rule that matches. At the bottom, there is always a catch-all rule that matches all messages. This rule can be modified using **Default action** in the **C-RUN Messages** window.

`*` is used as a wildcard.

See also *Using C-RUN, page 266*.

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.

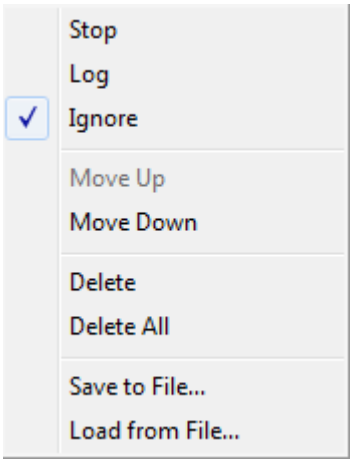
Source File The name of the source file and possibly the location in the file to match.

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 295*.
Load from File	Opens a dialog box where you can choose to load rules from a file.

COMPILER AND LINKER REFERENCE FOR C-RUN

--bounds_table_size

Syntax

```
--bounds_table_size records[:buckets]|(bytes)
```

Parameters

records	The number of records.
:*buckets*	The number of buckets.
(*bytes*)	The number of bytes, within parentheses.

For use with

The linker.

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

Detecting accesses outside the bounds of arrays and other objects, page 273.



Project>Options>Runtime Checking>Number of entries

--debug_heap

Syntax

```
--debug_heap
```

For use with

The linker.

Description

Use this linker option to use the checked heap.

See also

The checked heap provided by the library, page 265.



Project>Options>Runtime Checking>Use checked heap

--generate_entries_without_bounds

Syntax

```
--generate_entries_without_bounds
```

For use with

The compiler.

Description

Use this compiler option to generate extra functions for use from non-instrumented code. This option requires that out-of-bounds checking is enabled.

See also

Detecting accesses outside the bounds of arrays and other objects, page 273.



Project>Options>Runtime Checking>Generate functions callable from non-instrumented code

--ignore_uninstrumented_pointers

Syntax

```
--ignore_uninstrumented_pointers
```

For use with

The compiler.

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 273.



Project>Options>Runtime Checking>Check pointers from non-instrumented functions

--ignore_uninstrumented_pointers

Syntax

```
--ignore_uninstrumented_pointers
```

For use with

The linker.

Description

Use this linker option to disable checking of accessing via pointers in memory for which no bounds have been set.

See also

Detecting accesses outside the bounds of arrays and other objects, page 273.



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.
`div_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.
`bounds`	Checks for accesses outside the bounds of arrays and other objects.
`bounds_no_checks`	Tracks pointer bounds, but performs no checks. See also `#pragma disable_check = bounds`.

For use with

The compiler

Description

Use this compiler option to enable runtime error checking.

See also

Introduction to runtime error checking, page 264.



To set related options, choose:

Project>Options>Runtime Checking

#pragma default_no_bounds

Syntax

```
#pragma default_no_bounds [=on|=off]
```

Parameters

`on`	Makes the default for all functions declared from this point be as if they were declared with `#pragma no_bounds`.
`off`	Turns off the default.

Description

Use this pragma directive to apply `#pragma no_bounds` to a whole set of functions, for example around a header file declaring the interface to unchecked code.

See also

Detecting accesses outside the bounds of arrays and other objects, page 273.

#pragma define_with_bounds

Syntax

```
#pragma define_with_bounds
```

Description

You can only use this pragma directive on a function that is declared with `#pragma no_bounds` (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

```
#pragma define_without_bounds
```

Description

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 `#pragma disable_check = bounds` is used).

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.

Example

```
/* 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);
  ...
}
```

#pragma disable_check

Syntax

```
#pragma disable_check = bounds
```

Parameters

`bounds`	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

```
#pragma generate_entry_without_bounds
```

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 `--ignore_uninstrumented_pointers`, the given bounds will not cause errors.



It is an error to use this pragma directive on a function where no such entry can be generated. This includes functions that take a variable number of arguments, and functions that take one or more function pointers to functions that take or return values that contain pointers.

It is not an error to use this pragma directive on a function that does not need such an entry (because it takes no pointers, or because it is declared with `#pragma no_bounds`). In this case, no extra entry is generated.

See also

Detecting accesses outside the bounds of arrays and other objects, page 273.

#pragma no_arith_checks

Syntax

```
#pragma no_arith_checks
```

Description

Use this pragma directive to specify that no C-RUN arithmetic checks will be performed in the function that follows.

#pragma no_bounds

Syntax

```
#pragma no_bounds
```

Description

Use this pragma directive to specify that the immediately following function is not instrumented for bounds checking. No extra hidden bounds parameters will be passed when this function is called, and it will not return bounds for pointers, if any, in its return value.

See also

Detecting accesses outside the bounds of arrays and other objects, page 273.

__as_get_base

Syntax

```
__as_get_base(ptr)
```

Parameters

ptr	A pointer.

Description

Use this operator to create a pointer of the same type as *ptr*, representing the base of the area pointed to by *ptr*.

Example

```
base = __as_get_base(my_ptr);
```

__as_get_bound

Syntax

```
__as_get_bound(ptr)
```

Parameters

ptr	A pointer.

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(my_ptr);
```

__as_make_bounds

Syntax

```
__as_make_bounds(ptr, number)

__as_make_bounds(ptr, base, bound)
```

Parameters

ptr	A pointer that has no bounds.
number	The number of elements.
base	The start of the object pointed to.
bound	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

--rtc_enable

Syntax

```
--rtc_enable
```

Note that this option must be placed before the `--backend` option on the command line.

For use with

`cspybat`

Description

Use this option to enable C-RUN run-time checking in `cspybat`. This option is automatically enabled if any of the other `-rtc_*` options are used.



This option is not available in the IDE.

--rtc_output

Syntax

```
--rtc_output file
```

Note that this option must be placed before the `--backend` option on the command line.

Parameters

file The file for output messages.

For use with

cspybat

Description

Use this option to specify to cspybat a file for the C-RUN message output, in text (filename extension txt) or XML (filename extension xml) format.



This option is not available in the IDE.

--rtc_raw_to_txt

Syntax

```
--rtc_raw_to_txt=file
```

Note that this option must be placed before the --backend option on the command line.

For use with

cspybat

Description

Use this option to make cspybat 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

filename 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 287 for information about **Save to File**.



This option is not available in the IDE.

Part 3. Advanced debugging

Part 3. Advanced debugging

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

- *Multicore debugging, page 298*
- *Interrupts, page 306*
- *C-SPY macros, page 328*
- *The C-SPY command line utility—cspybat, page 407*
- *Flash loaders, page 459*

Multicore debugging

Contents

Introduction to multicore debugging 298
Briefly about multicore debugging 298
Symmetric multicore debugging 298
Asymmetric multicore debugging 299
Requirements and restrictions for multicore debugging 299
Debugging multiple cores 299
Setting up for symmetric multicore debugging 299
Setting up for asymmetric multicore debugging 300
Starting and stopping a multicore debug session 301
Reference information on multicore debugging 302
Cores window 302
Multicore toolbar 304
The multicore session file 304

INTRODUCTION TO 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 that run the same application. This is handled using a single instance of the IAR Embedded Workbench IDE.
- *Asymmetric multicore debugging* (AMP), which means debugging two or more cores that run separate applications. This is handled using two or more cooperating instances of the IAR Embedded Workbench IDE, where each instance is connected to one or more identical cores.

Symmetric multicore debugging

Symmetric multicore debugging means that the target has two or more identical cores that run the same application. The cores can typically 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 **Multicore** toolbar is a complement to the **Cores** window,
- The **Stack** window can show the stack for each core by means of dedicated stack sections.
- RTOS support is available in separate multicore-aware plugins, Typically, they work like their single-core plugin counterparts, but handle multiple active tasks on separate cores. The plugins might also provide the information required by the **Stack** window to display the stack for any selected task.

Asymmetric multicore debugging

Asymmetric multicore means that the target has two or more cores that run separate applications. To debug the target, two or more IDE instances can be used, where each instance is connected to one or more identical cores. The IDE instances synchronize so that debugging sessions can be started and stopped, and the cores can be controlled from any of the instances. Except for shared memory, each debugging session can only show information (variables, call stack, etc) about its own cores.

You start one IDE instance manually and that instance is referred to as the *master*. When you start an asymmetric multicore debugging session, the *master* instance can initiate one or more *partner* (or *slave*) instances. The partner instances will be reused if they are already running.

All instances each require their own project, master and partners. You must set up each project with the correct processor variant, linker, and debugger options. The master project must also be configured to act as multicore master or have multicore master mode enabled.

One possible strategy for download is to combine the debug images for the cores into one and let the master project download the combined image. In this scenario, the partners must be configured to attach to a running target, and/or to suppress any downloading.

Another strategy is to download the master and partners as separate binary images, in which case you must make sure to avoid any unintentional overlaps in memory.

This is an overview of special support for asymmetric multicore debugging:

- You can control whether to automatically start and stop the whole application or to run the cores independently of each other.
- Each instance of the IDE displays debug information for the cores that it is connected to.
- The **Cores** window shows a list of all available cores, and gives some information about each core, such as its execution state. The **Multicore** toolbar is a complement to the **Cores** window,
- When you set a breakpoint, it is only connected to one core, and when the breakpoint is triggered, that core is stopped.

Requirements and restrictions for multicore debugging

The C-SPY simulator supports multicore debugging for most cores and devices. If supported, there are no further specific requirements or restrictions.

To use multicore debugging in your hardware debugger system, you need a specific combination of C-SPY driver and debug probe:

- The IAR C-SPY I-jet driver
- The IAR C-SPY CMSIS-DAP driver
- The IAR C-SPY J-Link/J-Trace driver (asymmetric multicore)



There might be restrictions in trace support due to limitations in the hardware you are using.

DEBUGGING MULTIPLE CORES

Setting up for symmetric multicore debugging

1. Choose **Project>Options>Debugger>Multicore** and specify the number of cores you have.
2. You can now start your debug session.

Setting up for asymmetric multicore debugging

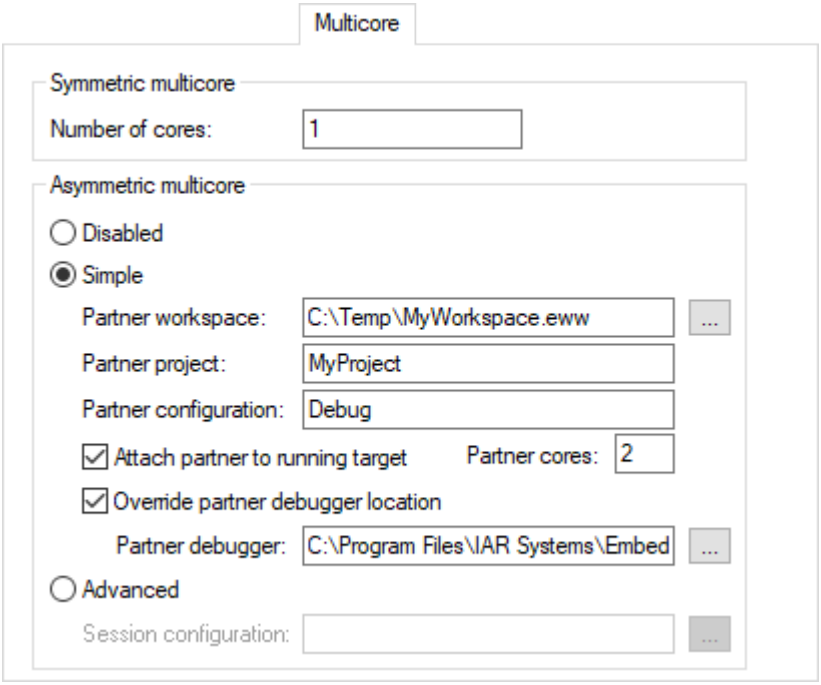
There are a number of ways that you can set up for multicore debugging, but this strategy is recommended:

1: Preparing the projects

1. Create a workspace with two or more projects, one for each core or set of cores.
2. Select an appropriate download strategy. One possible download strategy would be to combine the images for the cores into one and let the master project download the combined image. In this scenario, the partners would have to be configured to attach to a running target, and/or to suppress any downloading.
3. In the intended master project, choose **Project>Options>Debugger>Multicore** to open the **Multicore** options page.

2A: Setting up one partner project

1. Select **Simple**. Specify the options **Partner workspace** (path), **Partner project** (project name), and **Partner configuration** (build configuration). These settings are used when the partner session starts.



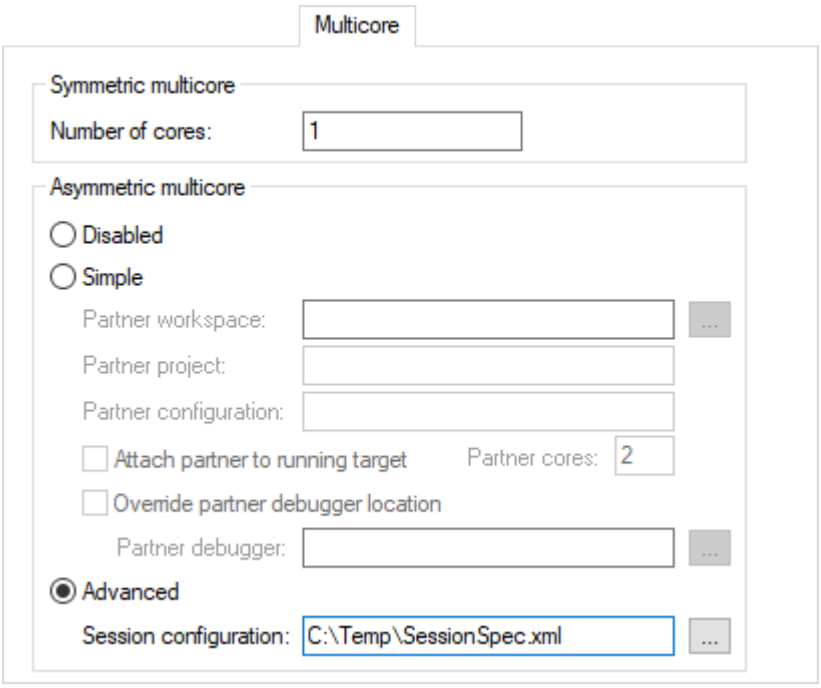
2. Select the option **Attach partner to running target**. Use the **Partner cores** option to specify the number of cores in the partner project.

   By default, the Embedded Workbench instance associated with the *partner* project must be installed in the same directory as the Embedded Workbench instance associated with the *master* project, for example in `c:\iar\ewarm-2.1w`. If the two Embedded Workbench instances were installed in *different* locations, you must select **Override partner debugger location** and specify the installation directory of the Embedded Workbench for the partner project. Note that the Embedded Workbench for the partner project must be based on version 9.1.7 or later of the shared components—to check this, choose **Help>About>Product Info**.

   For more information about the multicore settings, see *Multicore, page 470*

2B: Setting up two or more partner projects

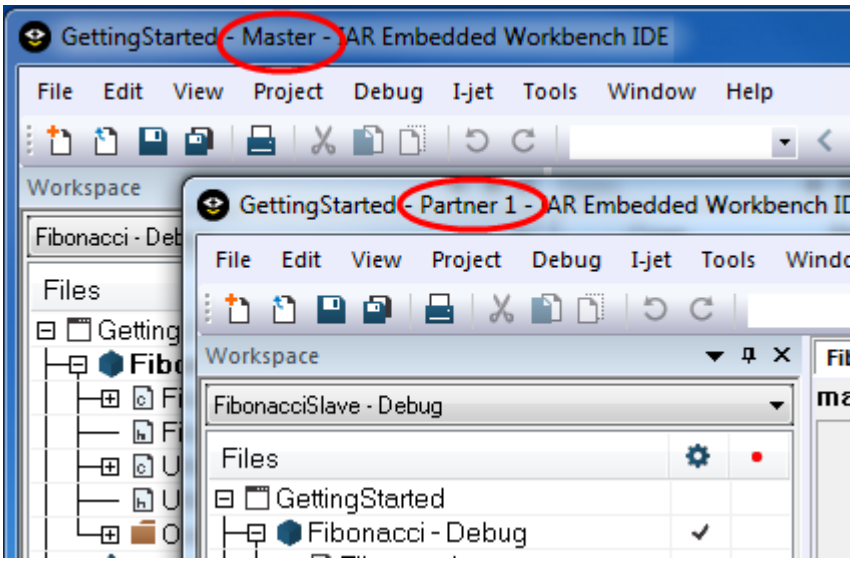
- Select **Advanced**. Create a multicore session file in XML format with information about location and settings of the partner projects and use the browse button to specify this file. For more information about this file, see *The multicore session file, page 304*. These settings are used when the partner session starts.



3: Make final settings

1. Select appropriate reset strategies for all projects:
   - In the master project, choose **Project>Options>*Driver*>Setup>Reset** and select a reset strategy, typically **Hardware**.
   - In the **Workspace** window, switch to the partner projects one at a time. Then for each project, choose **Project>Options>*Driver*>Setup>Reset** and select a reset strategy *for the partner session* that does not affect the master session, typically **Software**.
2. Make sure to use compatible settings for the debug probe for all projects.

The master and partner instances are indicated in the main IDE window title bar.



Starting and stopping a multicore debug session

1. To start a multicore debug session, for example use the standard **Download and Debug** command, either in the master or in a partner session.

2. To stop a multicore debug session, for example use the standard **Stop Debugging** command, which will stop all debugging sessions.

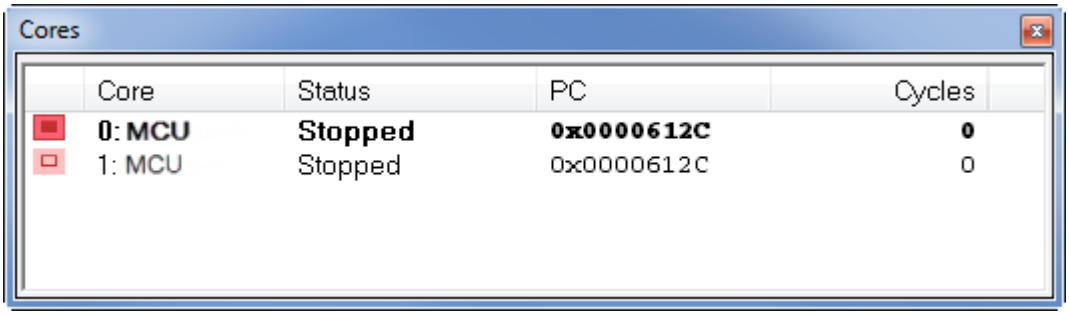
REFERENCE INFORMATION ON MULTICORE DEBUGGING

See also:

- *__getNumberOfCores, page 356*
- *__getSelectedCore, page 356*
- *__selectCore, page 379*

Cores window

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



This window shows a list of all available cores, and gives some information about each core, such as its execution state. The line highlighted in bold is the core currently in focus, which means that any window showing information that is specific to a core will be updated to reflect the state of the core in focus. This includes highlights in editor windows and the **Disassembly**, **Registers**, **Watch**, **Locals**, **Call Stack** window, and so on. Double-click a line to focus on that core.



For asymmetric multicore debugging, only local cores 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.

See also *Debugging multiple cores, page 299*.

Requirements

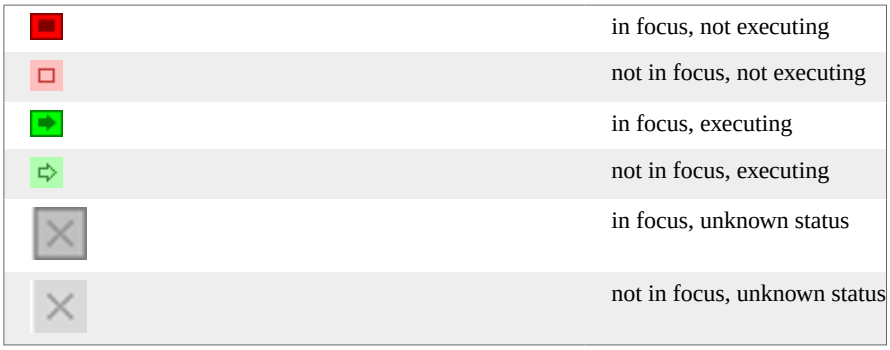
One of these alternatives:

- An I-jet or I-jet Trace debug probe
- A J-Link or J-Trace debug probe
- The C-SPY simulator. (Not for all cores and devices.)

Display area

A row in this area shows information about a core, in these columns:

Execution state Displays one of these icons to indicate the execution state of the core:



Core The name of the core.

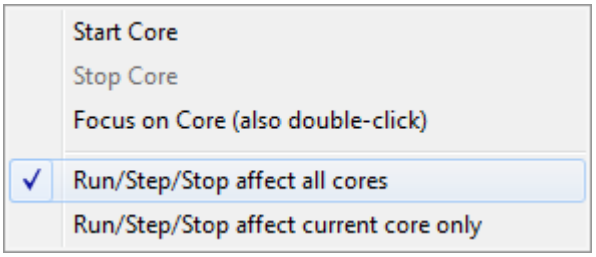
Status The status of the execution, which can be one of **Stopped**, **Running**, **Sleeping**, or **Unknown**.

PC The value of the program counter.

Cycles | Time The value of the cycle counter or the execution time since the start of the execution, depending on the debugger driver you are using.

Context menu

For symmetric multicore debugging, this context menu is available:



These commands are available:

Start Core Starts the selected core.

Stop Core Stops the selected core.

Focus on Core (also double-click) Focuses on the selected core.

Run/Step/Stop affect all cores The **Run**, **Step**, **Stop** commands affect all cores.

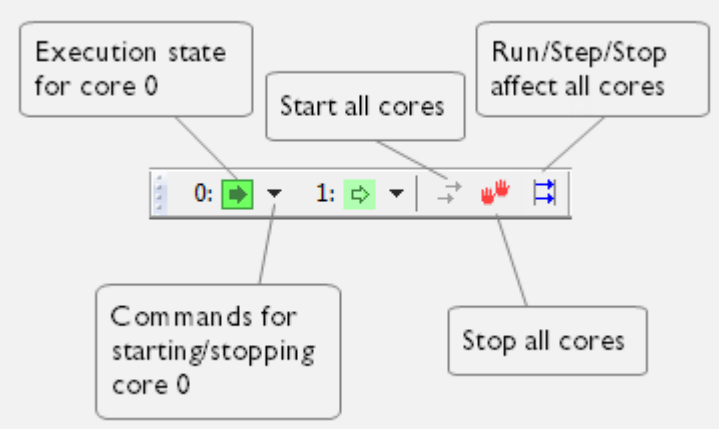
Run/Step/Stop affect current core only The **Run**/**Step**/**Stop** commands only affect the current core. This menu command is only supported if your device supports it.



These commands are not supported by all target hardware.

Multicore toolbar

The **Multicore** toolbar can be toggled on/off from the **Window>Toolbars** submenu when you have enabled multicore debugging, see *Setting up for symmetric multicore debugging, page 299* or *Setting up for asymmetric multicore debugging, page 300*, 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.



For asymmetric multicore debugging, you can use the toolbar commands to start and stop cores in the associated debugging session.

The multicore session file

This file in XML format can be used to specify an asymmetric debug session with more than two IDE instances. You specify the location of the file to the IDE on the **Project>Options>Debugger>Multicore** page. For more information, see *Setting up for asymmetric multicore debugging, page 300*, and *Multicore, page 470*,

XML specification

The multicore session file must look like this:

```
<?xml version="1.0" encoding="utf-8"?>

<sessionSetup>

    <partner>
        <name>Name_of_master_instance</name>
        <workspace>Path_to_workspace</workspace>
        <project>Name_of_project</project>
        <config>Build_config</config>
        <numberOfCores>N</numberOfCores>
        <attachToRunningTarget>true/false</attachToRunningTarget>
    </partner>

    <partner>
        <name>Name_of_partner_instance_1</name>
        <workspace>Path_to_workspace</workspace>
        <project>Name_of_project</project>
        <config>Build_config</config>
        <numberOfCores>N</numberOfCores>
        <attachToRunningTarget>true/false</attachToRunningTarget>

        <debuggerpath>Path_to_Embedded_workbench</debuggerpath>
    </partner>

    ...

     <partner>
        <name>Name_of_partner_instance_N</name>
        <workspace>Path_to_workspace</workspace>
        <project>Name_of_project</project>
        <config>Build_config</config>
        <numberOfCores>N</numberOfCores>
        <attachToRunningTarget>true/false</attachToRunningTarget>

        <debuggerpath>Path_to_Embedded_workbench</debuggerpath>
    </partner>

</sessionSetup>
```

Interrupts

Contents

Introduction to interrupts 306
Briefly about the interrupt simulation system 306
Interrupt characteristics 307
Interrupt simulation states 308
C-SPY system macros for interrupt simulation 308
Target-adapting the interrupt simulation system 309
Briefly about interrupt logging 309
Using the interrupt system 309
Simulating a simple interrupt 310
Simulating an interrupt in a multi-task system 311
Getting started using interrupt logging 311
Reference information on interrupts 312
Interrupt Configuration window 312
Available Interrupts window 314
Interrupt Status window 316
Interrupt Log window 318
Interrupt Log Summary window 321
Timeline window—Interrupt Log graph 324

INTRODUCTION TO INTERRUPTS

See also:

- *Reference information on C-SPY system macros, page 343*
- *Breakpoints, page 90*
- The *IAR C/C++ Development Guide for Arm*

Briefly about the interrupt simulation system

By simulating interrupts, you can test the logic of your interrupt service routines and debug the interrupt handling in the target system long before any hardware is available. If you use simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices.

For most cores and 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. Note that for some cores and devices, the C-SPY simulator uses the Imperas Instruction Set Simulator instead of the IAR C-SPY simulator driver, see *The IAR C-SPY Simulator, page 28*. The Imperas Instruction Set Simulator does not support simulated interrupts. When C-SPY uses the Imperas driver, the interrupt system more resembles a 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.

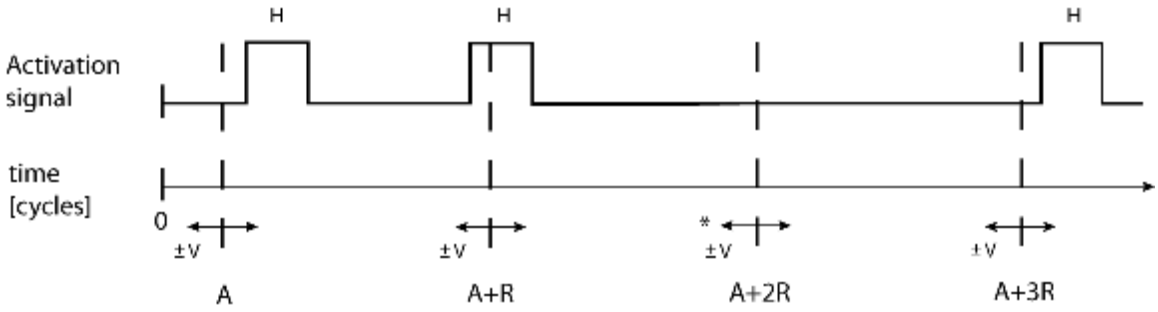
All interrupts you define using the **Interrupt Configuration** window are preserved between debug sessions, unless you remove them. A forced interrupt, on the other hand, exists only until it has been serviced and is not preserved between sessions.



The interrupt simulation system is activated by default, but if not required, you can turn off the interrupt simulation system to speed up the simulation. To turn it off, use either the **Interrupt Configuration** window or a system macro.

Interrupt characteristics

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability*.



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

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

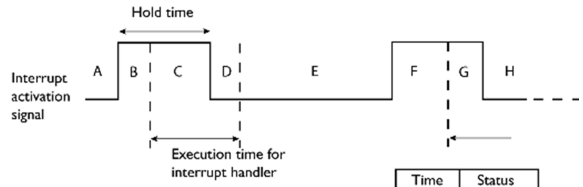
The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation time. However, interrupts can only be raised between instructions, which means that a full assembler instruction must have been executed before the interrupt is generated, regardless of how many cycles an instruction takes.

To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—the probability, in percent, that the interrupt will actually appear in a period—and *variance*—a time variation range as a percentage of the repeat interval. These options make it possible to randomize the interrupt simulation. You can also specify a *hold time* which describes how long the interrupt remains pending until removed if it has not been processed. If the hold time is set to *infinite*, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

Interrupt simulation states

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

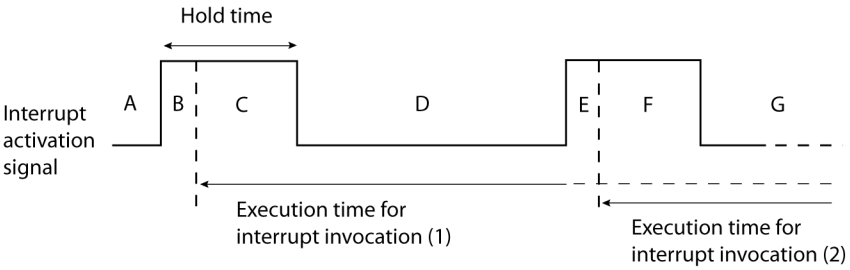
Normally, a repeatable interrupt has a specified repeat interval that is longer than the execution time. In this case, the status information at different times looks like this:



Time	Status
A	Idle
B	Pending
C, D	Executing
E	Idle
F	Pending
G, H	Executing

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:



Time	Status
A	Idle
B	Pending
C, D, E	Executing
F, G	1st interrupt: Suspended 2nd interrupt: Executing

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.

When the IAR C-SPY simulator driver is used (see *Briefly about the interrupt simulation system, page 306*), the C-SPY Simulator provides these predefined system macros related to interrupts:

- `__cancelAllInterrupts`
- `__cancelInterrupt`

- `__disableInterrupts`
- `__enableInterrupts`
- `__orderInterrupt`
- `__popSimulatorInterruptExecutingStack`

The parameters of the first five macros correspond to the equivalent entries of the **Interrupt Configuration** window.

For more information about each macro, see *Reference information on C-SPY system macros, page 343*.

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 35*.

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 driver (see *Briefly about the interrupt simulation system, page 306*), you can also log internal interrupt status information, such as triggered, expired, etc. In the IDE:

- The logs are displayed in the **Interrupt Log** window
- A summary is available in the **Interrupt Log Summary** window
- The Interrupt graph in the **Timeline** window provides a graphical view of the interrupt events during the execution of your application

Requirements for interrupt logging

Interrupt logging is supported by the C-SPY simulator driver (see *Briefly about the interrupt simulation system, page 306*). The Imperas Instruction Set Simulator does not support interrupt logging.

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, 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
- A TI XDS debug probe and an SWD interface between the debug probe and the target system

See also *Getting started using interrupt logging, page 311*.

USING THE INTERRUPT SYSTEM

See also:

- *Using C-SPY macros, page 332* for details about how to use a setup file to define simulated interrupts at C-SPY startup
- The tutorial *Simulating an interrupt* in the Information Center.

Simulating a simple interrupt

This example demonstrates the method for simulating a timer interrupt. However, the procedure can also be used for other types of interrupts.

To simulate and debug an interrupt:

1. Assume this simple application which contains an 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 <stdio.h>

unsigned int ticks = 0;

/* IRQ handler */
#if __ARM_PROFILE_M__
/* Defines an interrupt handler for the Cortex-M UART interrupt. */
void UART_Handler()
#else
/* Defines an interrupt handler for other cores. */
__irq __arm void IRQ_Handler(void)
#endif

{
  /* 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 Configuration** to open the **Interrupt Configuration** window. Right-click in the window and select **Enable Interrupt Simulation** on the context menu. For the timer example, verify these settings:

Option	Settings
Interrupt	IRQ
First activation	4000
Repeat interval	2000
Hold time	10
Probability (%)	100

Option	Settings
Variance (%)	0

Table 14. 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—Interrupt Log graph, page 324*.

Simulating an interrupt in a multi-task system

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

To simulate a normal interrupt exit:

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

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

Getting started using interrupt logging

1. To set up for interrupt logging, choose ***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>Communication** page and the **Project>Options>TI XDS>Communication** page, respectively.

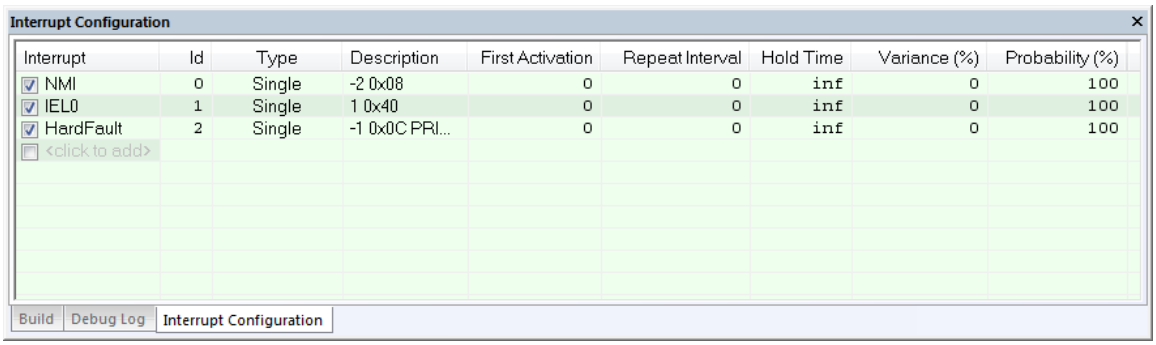
   For the C-SPY simulator, no specific settings are required.
2. Choose ***Driver*>Interrupt Log** to open the **Interrupt Log** window. Optionally, you can also choose:
   - ***Driver*>Interrupt Log Summary** to open the **Interrupt Log Summary** window
   - ***Driver*>Timeline** to open the **Timeline** 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 the **Interrupt Log** or **Interrupt Log Summary** window, 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

Interrupt Configuration window

The **Interrupt Configuration** window is available by choosing **Simulator>Interrupt Configuration**.



This window lists all installed interrupts. Use this window to enable or disable individual interrupts or the interrupt simulation system, and to edit the properties of installed interrupts.

See also *Using the interrupt system, page 309*.

Requirements

The C-SPY simulator. Not available for all cores and devices.

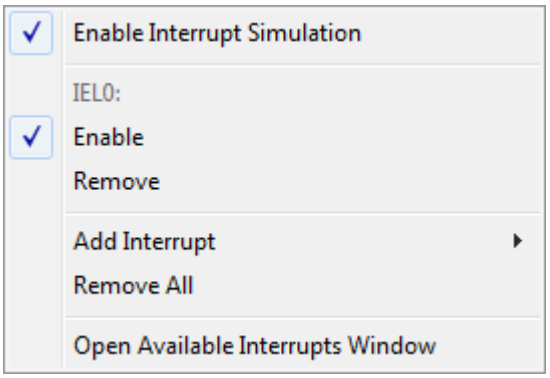
Display area

This area contains these columns:

Interrupt	Lists all installed interrupts. Use the checkbox to enable or disable the interrupt.
ID	A unique interrupt identifier.
Type	Shows the type of the interrupt. The type can be one of: **Forced**, a single-occasion interrupt defined in the **Available Interrupts** window. **Single**, a single-occasion interrupt. **Repeat**, a periodically occurring interrupt. If the interrupt has been set from a C-SPY macro, the additional part **(macro)** is added, for example: **Repeat(macro)**.
Description	A description of the selected interrupt, if available. The description is retrieved from the selected device description file and consists of a string describing the 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	The value of the cycle counter after which the specified interrupt will be generated. Click to edit.
Repeat Interval	The periodicity of the interrupt in cycles. Click to edit.
Hold Time	How long, in cycles, the interrupt remains pending until removed if it has not been processed. Click to edit. If you specify `inf`, the corresponding pending bit will be set until the interrupt is acknowledged or removed.
Variance %	A timing variation range, as a percentage of the repeat interval, in which the interrupt might occur for a period. For example, if the repeat interval is 100 and the variance 5%, the interrupt might occur anywhere between T=95 and T=105, to simulate a variation in the timing. Click to edit.
Probability %	The probability, in percent, that the interrupt will actually occur within the specified period. Click to edit.

Context menu

This context menu is available:

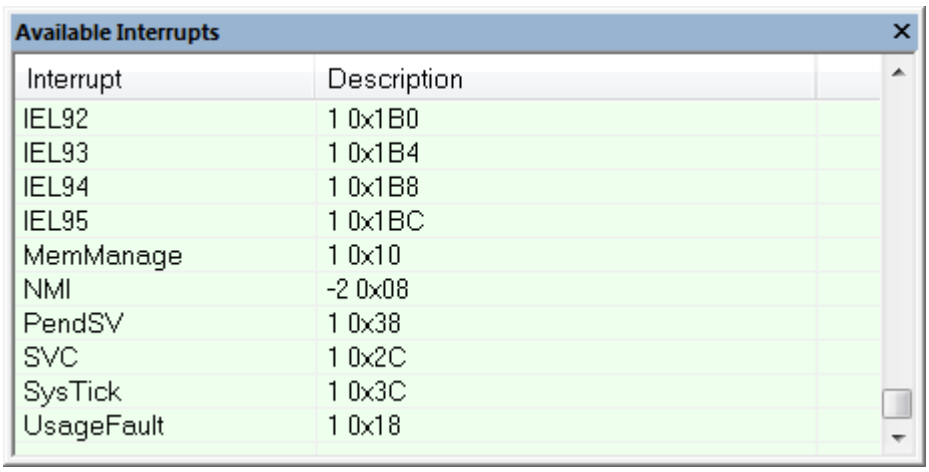


These commands are available:

Enable Interrupt Simulation	Enables or disables the entire interrupt simulation system. If the interrupt simulation is disabled, the definitions remain but no interrupts are generated. Note that you can also enable and disable installed interrupts individually by using the check box to the left of the interrupt name in the list of installed interrupts.
Enable	Enables or disables the individual interrupt you clicked on.
Remove	Removes the individual interrupt you clicked on.
Add Interrupt	Selects an interrupt to install. The drop-down list contains all available interrupts. Your selection will automatically update the **Description** box. The list is, 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.
Remove All	Removes all installed interrupts in the window.
Open Available Interrupts Window	Opens the **Available Interrupts** window, see *Available Interrupts window, page 314*.

Available Interrupts window

The **Available Interrupts** window is available from the C-SPY driver menu.



Use this window for an overview of all available interrupts for your project. You can also use it for forcing an interrupt instantly. This is useful when you want to check your interrupt logic and interrupt routines. Just start typing an interrupt name and focus shifts to the first line found with that name.

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

To sort the window contents, click on either the **Interrupt** or the **Description** column header. A second click on the same column header reverses the sort order.

To force an interrupt:

1. Enable the interrupt simulation system, see *Interrupt Configuration window, page 312*.
2. Activate the interrupt by choosing the **Force Interrupt** command from the context menu.

Requirements

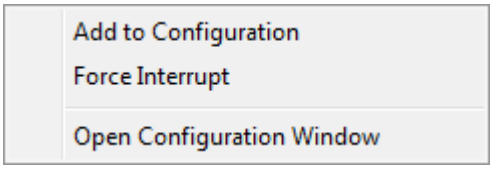
The C-SPY simulator. Not available for all cores and devices.

Display area

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

Context menu

This context menu is available:

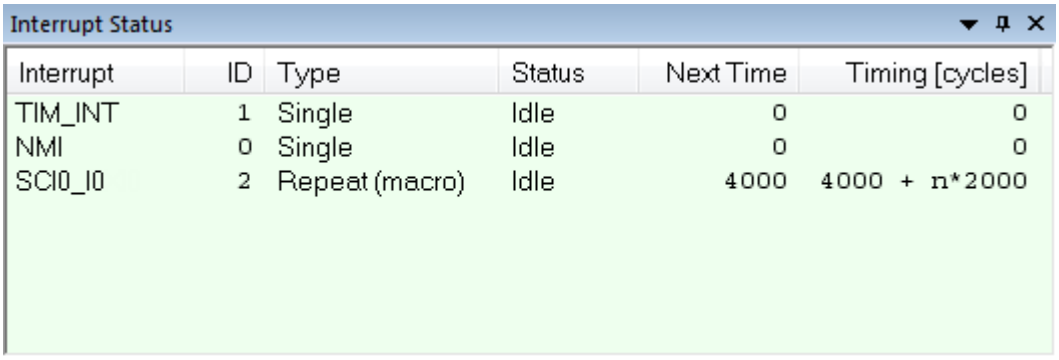


These commands are available:

Add to Configuration	Installs the selected interrupt and adds it to the **Interrupt Configuration** window.
Force Interrupt	Triggers the selected interrupt.
Open Configuration Window	Opens the **Interrupt Configuration** window, see *Interrupt Configuration window, page 312*.

Interrupt Status window

The **Interrupt Status** window is available from the C-SPY driver menu.



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. Not available for all cores and devices.

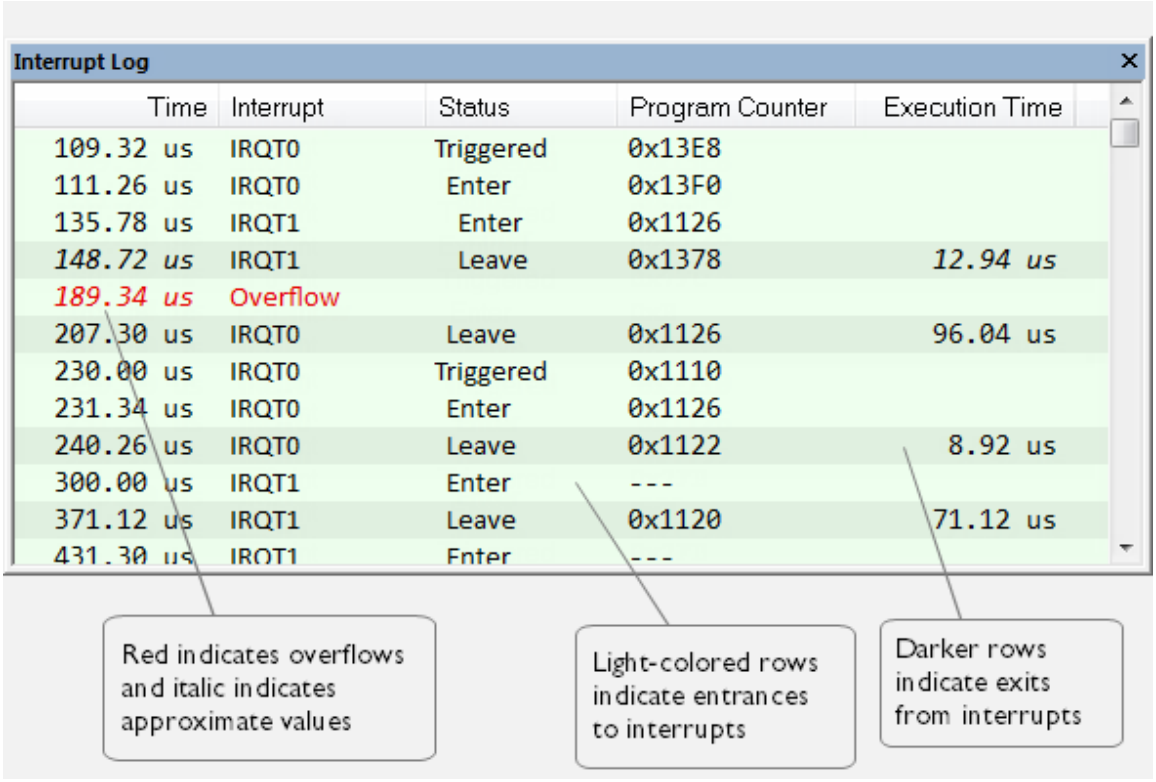
Display area

This area contains these columns:

Interrupt	Lists all interrupts.
ID	A unique interrupt identifier.
Type	The type of the interrupt. The type can be one of: **Forced**, a single-occasion interrupt defined in the **Available Interrupts** window. **Single**, a single-occasion interrupt. **Repeat**, a periodically occurring interrupt. If the interrupt has been set from a C-SPY macro, the additional part **(macro)** is added, for example: **Repeat(macro)**.
Status	The state of the interrupt: **Idle**, the interrupt activation signal is low (deactivated). **Pending**, the interrupt activation signal is active, but the interrupt has not been yet acknowledged by the interrupt handler. **Executing**, the interrupt is currently being serviced, that is the interrupt handler function is executing. **Suspended**, the interrupt is currently suspended due to execution of an interrupt with a higher priority. **(deleted)** is added to **Executing** and **Suspended** if you have deleted a currently active interrupt. **(deleted)** is removed when the interrupt has finished executing.
Next Time	The next time an idle interrupt is triggered. Once a repeatable interrupt stats executing, a copy of the interrupt will appear with the state Idle and the next time set. For interrupts that do not have a next time—that is pending, executing, or suspended—the column will show `--`.
Timing	The timing of the interrupt. For a **Single** and **Forced** interrupt, the activation time is displayed. For a **Repeat** interrupt, the information has the form: `Activation Time + n*Repeat Time`. For example, `2000 + n*2345`. This means that the first time this interrupt is triggered, is at 2000 cycles and after that with an interval of 2345 cycles.

Interrupt Log window

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



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.



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

For more information, see *Getting started using interrupt logging, page 311*.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window—Interrupt Log graph, page 324*.

Requirements

One of these alternatives:

- The C-SPY simulator. Not available for all cores and devices.
- An I-jet or I-jet Trace in-circuit debugging 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
- A TI XDS debug probe and 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 approximate 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).

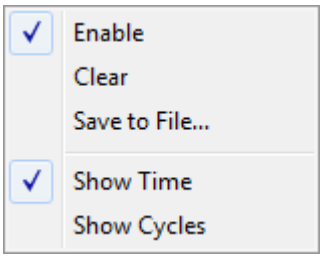
Display area for the C-SPY simulator

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 **Available Interrupts** window. **Enter**, the interrupt is currently executing. **Leave**, the interrupt has been executed. **Expired**, the interrupt hold time has expired without the interrupt being executed. **Rejected**, the interrupt has been rejected because the necessary interrupt registers were not set up to accept the interrupt.
Program Counter	The value of the program counter when the event occurred.
Execution Time/ Cycles	The time spent in the interrupt, calculated using the Enter and Leave timestamps. This includes time spent in any subroutines or other interrupts that occurred in the specific interrupt.

Context menu

This context menu is available:

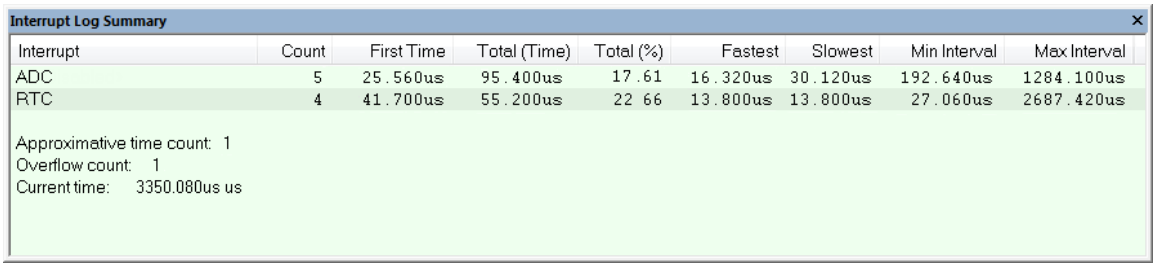


These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Save to File	Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Interrupt Log Summary window

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



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

For more information, see *Getting started using interrupt logging, page 311*.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window—Interrupt Log graph, page 324*.

Requirements

One of these alternatives:

- The C-SPY simulator. Not available for all cores and devices.
- An I-jet or I-jet Trace in-circuit debugging 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
- A TI XDS debug probe and an SWD interface between the debug probe and the target system

Display area for the C-SPY simulator

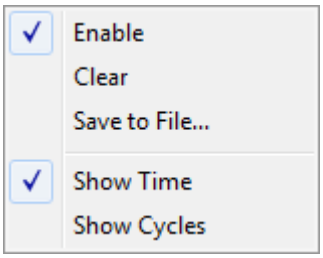
Each row in this area displays statistics about the specific interrupt based on the log information in these columns; and summary information is listed at the bottom of the display area:

Interrupt	The type of interrupt that occurred.
Count	The number of times the interrupt occurred.
First Time	The first time the interrupt was executed.
Total (Time)**	The accumulated time spent in the interrupt.
Total (%)	The time in percent of the current time.
Fastest**	The fastest execution of a single interrupt of this type.
Slowest**	The slowest execution of a single interrupt of this type.
Min Interval	The shortest time between two interrupts of this type. The interval is specified as the time interval between the entry time for two consecutive interrupts.
Max Interval	The longest time between two interrupts of this type. The interval is specified as the time interval between the entry time for two consecutive interrupts.
Approximative time count	The information displayed depends on the C-SPY driver you are using. For some C-SPY drivers, this information is not displayed or the value is always zero. In this case, all logs have an exact time stamp. For other C-SPY drivers, a non-zero value is displayed. The value represents the amount of logs with an approximative time stamp. This might happen if the bandwidth in the communication channel is too low compared to the amount of data packets generated by the CPU or if the CPU generated packets with an approximative time stamp.
Overflow count	The information displayed depends on the C-SPY driver you are using. For some C-SPY drivers, this information is not displayed or the value is always zero. For other C-SPY drivers, the number represents the amount of overflows in the communication channel which can cause logs to be lost. If this happens, it indicates that logs might be incomplete. To solve this, make sure not to use all C-SPY log features simultaneously or check used bandwidth for the communication channel.
Current time/Current cycles	The information displayed depends on the C-SPY driver you are using. For some C-SPY drivers, the value is always zero or not visible at all. For other C-SPY drivers, the number represents the current time or cycles—the number of cycles or the execution time since the start of execution.

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

Context menu

This context menu is available:

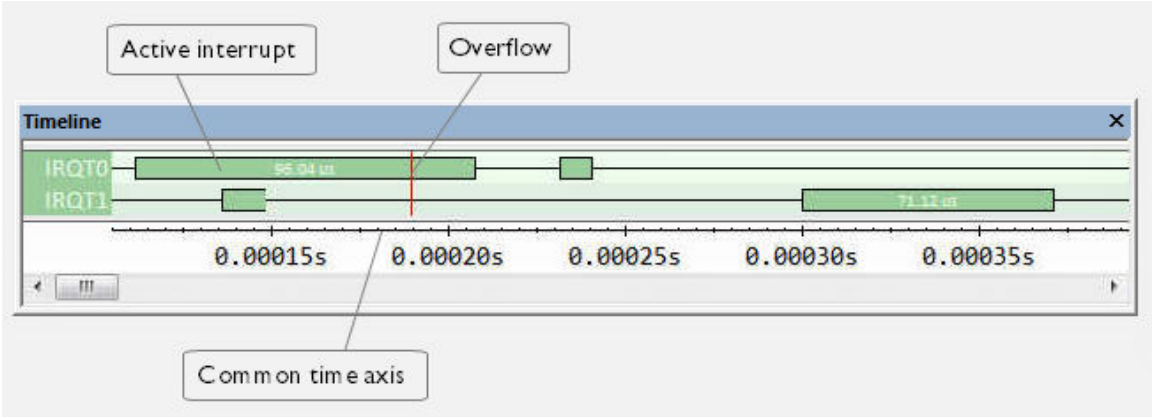


These commands are available:

Enable	Enables the logging system. The system will log information also when the window is closed.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Save to File	Displays a standard file selection dialog box where you can select the destination file for the log information. The entries in the log file are separated by `TAB` and `LF` characters. An **X** in the **Approx** column indicates that the timestamp is an approximation.
Show Time	Displays the **Time** column. If the **Time** column is displayed by default in the C-SPY driver you are using, this menu command is not available.
Show Cycles	Displays the **Cycles** column. If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Timeline window—Interrupt Log graph

The Interrupt Log graph displays interrupts collected by the trace system. In other words, the graph provides a graphical view of the interrupt events during the execution of your application.



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

Requirements

One of these alternatives:

- The C-SPY simulator. Not available for all cores and devices.
- An I-jet or I-jet Trace in-circuit debugging 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
- A TI XDS debug probe and an SWD interface between the debug probe and the target system

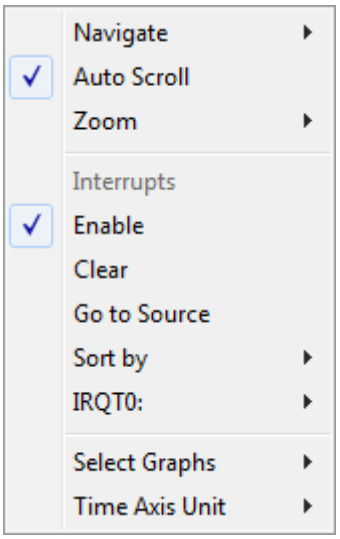
Display area

- The label area at the left end of the graph displays the names of the interrupts.
- The graph itself shows active interrupts as a thick green horizontal bar where the white figure indicates the time spent in the interrupt. This graph is a graphical representation of the information in the **Interrupt Log** window, see *Interrupt Log window, page 318*.
- If the bar is displayed without horizontal borders, there are two possible causes:
  - The interrupt is reentrant and has interrupted itself. Only the innermost interrupt will have borders.
  - There are irregularities in the interrupt enter-leave sequence, probably due to missing logs.
- If the bar is displayed without a vertical border, the missing border indicates an approximate time for the log.
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all interrupt logs from the target system.

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

Context menu

This context menu is available:



The exact contents of the context menu you see on the screen depends on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Navigate	Commands for navigating the graph(s). Choose between: **Next** moves the selection to the next relevant point in the graph. Shortcut key: right arrow. **Previous** moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow. **First** moves the selection to the first data entry in the graph. Shortcut key: Home. **Last** moves the selection to the last data entry in the graph. Shortcut key: End. **End** moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.
Auto Scroll	Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.
Zoom	Commands for zooming the window, in other words, changing the time scale. Choose between: **Zoom to Selection** makes the current selection fit the window. Shortcut key: Return. **Zoom In** zooms in on the time scale. Shortcut key: + **Zoom Out** zooms out on the time scale. Shortcut key: – **10ns**, **100ns**, **1us**, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window. **1ms**, **10ms**, etc makes an interval of 1 millisecond or 10 milliseconds, respectively, fit the window. **10m**, **1h**, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.
Interrupts	A heading that shows that the Interrupt Log-specific commands below are available.

Enable	Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.
Clear	Deletes the log information. Note that this will also happen when you reset the debugger.
Go To Source	Displays the corresponding source code in an editor window, if applicable.
Sort by	Sorts the entries according to their ID or name. The selected order is used in the graph when new interrupts appear.
source	Goes to the previous/next log for the selected source.
Select Graphs	Selects which graphs to be displayed in the **Timeline** window.
Time Axis Unit	Selects the unit used in the time axis—choose between **Seconds** and **Cycles**. If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.

C-SPY macros

Contents

Introduction to C-SPY macros ... 330
Reasons for using C-SPY macros ... 330
Briefly about using C-SPY macros ... 330
Briefly about setup macro functions and files ... 331
Briefly about the macro language ... 331
Using C-SPY macros ... 332
Registering C-SPY macros—an overview ... 332
Executing C-SPY macros—an overview ... 332
Registering and executing using setup macros and setup files ... 332
Executing macros using Quick Watch ... 333
Executing a macro by connecting it to a breakpoint ... 334
Aborting a C-SPY macro ... 334
Reference information on the macro language ... 335
Macro functions ... 335
Macro variables ... 335
Macro parameters ... 335
Macro strings ... 336
Macro statements ... 336
Formatted output ... 337
Reference information on reserved setup macro function names ... 338
execConfigureTraceETM ... 339
execConfigureTraceSWO ... 339
execUserAttach ... 339
execUserPreload ... 339
execUserExecutionStarted ... 340
execUserExecutionStopped ... 340
execUserFlashInit ... 340
execUserSetup ... 340
execUserFlashReset ... 341
execUserPreReset ... 341
execUserReset ... 341
execUserStatus ... 342
execUserExit ... 342
execUserFlashExit ... 342
execUserCoreConnect ... 343
Reference information on C-SPY system macros ... 343
Summary of C-SPY system macros ... 343
__abortLaunch ... 347
__cancelAllInterrupts ... 347
__cancelInterrupt ... 347
__clearBreak ... 348
__closeFile ... 348
__delay ... 349
__disableInterrupts ... 349
__driverType ... 349
__emulatorSpeed ... 350
__emulatorStatusCheckOnRead ... 351
__enableInterrupts ... 351
__evaluate ... 352
__expandVar ... 352

__fillMemory8 ... 353
__fillMemory16 ... 354
__fillMemory32 ... 354
__fillMemory64 ... 355
__gdbserver_exec_command ... 356
__getNumberOfCores ... 356
__getSelectedCore ... 356
__getTracePortSize ... 357
__hasDAPRegs ... 358
__hwJetResetWithStrategy ... 358
__hwReset ... 359
__hwResetRunToBp ... 359
__hwResetWithStrategy ... 360
__hwRunToBreakpoint ... 361
__isBatchMode ... 362
__isMacroSymbolDefined ... 362
__jlinkExecCommand ... 363
__jlinkExecMacro ... 363
__jtagCommand ... 364
__jtagCP15IsPresent ... 364
__jtagCP15ReadReg ... 365
__jtagCP15WriteReg ... 365
__jtagData ... 365
__jtagRawRead ... 366
__jtagRawSync ... 366
__jtagRawWrite ... 367
__jtagResetTRST ... 368
__loadImage ... 368
__memoryRestore ... 369
__memorySave ... 370
__messageBoxYesCancel ... 370
__messageBoxYesNo ... 371
__openFile ... 371
__orderInterrupt ... 372
__popSimulatorInterruptExecutingStack ... 373
__probeType ... 374
__readAPReg ... 374
__readDPReg ... 375
__readFile ... 375
__readFileByte ... 376
__readMemory8, __readMemoryByte ... 376
__readMemory16 ... 377
__readMemory32 ... 377
__readMemory64 ... 377
__registerMacroFile ... 378
__resetFile ... 378
__restoreSoftwareBreakpoints ... 379
__selectCore ... 379
__setCodeBreak ... 380
__setDataBreak ... 381
__setDataLogBreak ... 383
__setLogBreak ... 384
__setSimBreak ... 385
__setSystemMaxData ... 385
__setSystemTimeout ... 386

__setTraceStartBreak ... 387
__setTraceStopBreak ... 388
__sourcePosition ... 389
__strFind ... 390
__subString ... 390
__system1 ... 391
__system2 ... 392
__system3 ... 392
__targetDebuggerVersion ... 393
__toLower ... 393
__toString ... 394
__toUpper ... 395
__unloadImage ... 395
__wallTime_ms ... 396
__whichCore ... 396
__writeAPReg ... 397
__writeDPReg ... 397
__writeFile ... 398
__writeFileByte ... 398
__writeMemory8, __writeMemoryByte ... 399
__writeMemory16 ... 399
__writeMemory32 ... 399
__writeMemory64 ... 400
Graphical environment for macros ... 401
Macro Registration window ... 401
Debugger Macros window ... 402
Macro Quicklaunch window ... 404

INTRODUCTION TO C-SPY MACROS

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, page 306*. 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 338*.

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 38*.

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 335*.

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;
 }
}
```



Reserved macro words begin with double underscores to prevent name conflicts.

USING C-SPY MACROS

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 38*

Registering C-SPY macros—an overview

C-SPY must know that you intend to use your defined macro functions, and therefore you must *register* your macros. There are various ways to register macro functions:

- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files, page 332*.
- You can register macros interactively in the **Macro Registration** window, see *Macro Registration window, page 401*. Registered macros appear in the **Debugger Macros** window, see *Debugger Macros window, page 402*.
- 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 378*.

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 332*.
- 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 333*.
- The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is more specifically designed for C-SPY macros. See *Macro Quicklaunch window, page 404*.
- 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 334*.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to execute your macro.

Registering and executing using setup macros and setup files

It can be convenient to register a macro file during the C-SPY startup sequence. To do this, specify a macro file which you load before starting the debug session. Your macro functions will be automatically registered each time you start the debugger.

If you use the reserved setup macro function names to define the macro functions, you can define exactly at which stage you want the macro function to be executed.

To define a setup macro function and load it during C-SPY startup:

1. Create a new text file where you can define your macro function.
   For example:

```
execUserSetup()
{
 ...
 __registerMacroFile("MyMacroUtils.mac");
 __registerMacroFile("MyDeviceSimulation.mac");

}
```

   This macro function registers the additional macro files `MyMacroUtils.mac` and `MyDeviceSimulation.mac`. Because the macro function is defined with the function name `execUserSetup`, it will be executed directly after your application has been downloaded.
2. Save the file using the filename extension `mac`.
3. Before you start C-SPY, choose **Project>Options>Debugger>Setup**. Select the **Use macro file** option, and choose the macro file you just created.
   The macros will now be registered during the C-SPY startup sequence.

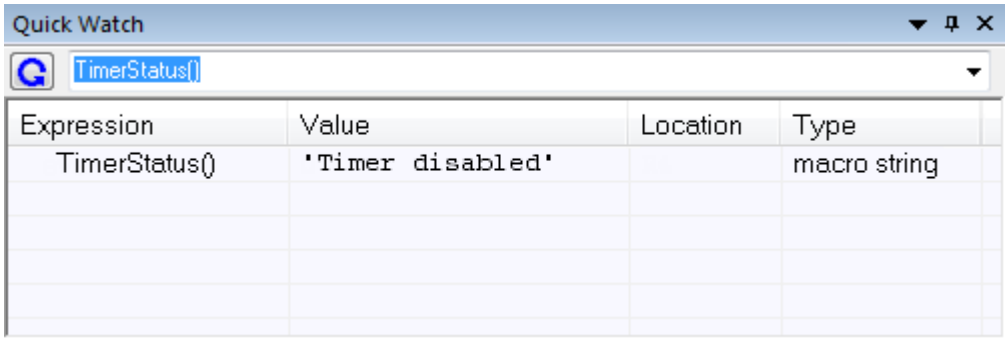
Executing macros using Quick Watch

The **Quick Watch** window lets you dynamically choose when to execute a macro function.

1. Consider this simple macro function that checks the status of a timer enable bit:

```
TimerStatus()
{
  if ((TimerStatreg & 0x01) != 0)/* Checks the status of reg */
    return "Timer enabled"; /* C-SPY macro string used */
  else
    return "Timer disabled"; /* C-SPY macro string used */
}
```

2. Save the macro function using the filename extension `mac`.
3. To load the macro file, choose **View>Macros>Macro Registration**. The **Macro Registration** window is displayed. Click **Add** and locate the file using the file browser. The macro file appears in the list of macros in the **Macro Registration** window.
4. Select the macro you want to register and your macro will appear in the **Debugger Macros** window.
5. Choose **View>Quick Watch** to open the **Quick Watch** window, type the macro call `TimerStatus()` in the text field and press Return,
   Alternatively, in the macro file editor window, select the macro function name `TimerStatus()`. Right-click, and choose **Quick Watch** from the context menu that appears.



   The macro will automatically be displayed in the **Quick Watch** window. For more information, see *Quick Watch window, page 84*.

Executing a macro by connecting it to a breakpoint

You can connect a macro to a breakpoint. The macro will then be executed when the breakpoint is triggered. The advantage is that you can stop the execution at locations of particular interest and perform specific actions there.



For instance, you can easily produce log reports containing information such as how the values of variables, symbols, or registers change. To do this you might set a breakpoint on a suspicious location and connect a log macro to the breakpoint. After the execution you can study how the values of the registers have changed.

To create a log macro and connect it to a breakpoint:

1. Assume this skeleton of a C function in your application source code:

```
int fact(int x)
{
 ...
}
```

2. Create a simple log macro function like this example:

```
logfact()
{
 __message "fact(" ,x, ")";
}
```

   The `__message` statement will log messages to the **Debug Log** window.

   Save the macro function in a macro file, with the filename extension `mac`.

3. To register the macro, choose **View**>**Macros**>**Macro Registration** to open the **Macro Registration** window and add your macro file to the list. Select the file to register it. Your macro function will appear in the **Debugger Macros** window.
4. To set a code breakpoint, click the **Toggle Breakpoint** button on the first statement within the function `fact` in your application source code. Choose **View**>**Breakpoints** to open the **Breakpoints** window. Select your breakpoint in the list of breakpoints and choose the **Edit** command from the context menu.
5. To connect the log macro function to the breakpoint, type the name of the macro function, `logfact()`, in the **Action** field and click **OK** to close the dialog box.
6. Execute your application source code. When the breakpoint is triggered, the macro function will be executed. You can see the result in the **Debug Log** window.

   Note that the expression in the **Action** field is evaluated only when the breakpoint causes the execution to really stop. If you want to log a value and then automatically continue execution, you can either:
   - Use a **Log** breakpoint, see *Log breakpoints dialog box, page 106*
   - Use the **Condition** field instead of the **Action** field. For an example, see  *Performing a task and continuing execution, page 99*.
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 337*.

   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

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 68*.

The syntax for defining one or more macro variables is:

```
__var nameList;
```

where *nameList* is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and it exists throughout the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and destroyed on return from the macro.

By default, macro variables are treated as signed integers and initialized to 0. When a C-SPY variable is assigned a value in an expression, it also acquires the type of that expression. For example:

Expression	What it means
`myvar = 3.5;`	`myvar` is now type `double`, value `3.5`.
`myvar = (int*)i;`	`myvar` is now type pointer to `int`, and the value is the same as `i`.

Table 15. Examples of C-SPY macro variables

In case of a name conflict between a C symbol and a C-SPY macro variable, C-SPY macro variables have a higher precedence than C variables. Note that macro variables are allocated on the debugger host and do not affect your application.

Macro parameters

A macro parameter is intended for parameterization of device support. The named parameter will behave as a normal C-SPY macro variable with these differences:

- The parameter definition can have an initializer
- Values of a parameters can be set through options (either in the IDE or in `cspybat`).
- A value set from an option will take precedence over a value set by an initializer

- A parameter must have an initializer, be set through an option, or both. Otherwise, it has an undefined value, and accessing it will cause a runtime error.

The syntax for defining one or more macro parameters is:

```
__param param[=value, ...;]
```

Use the command line option `--macro_param` to specify a value to a parameter, see *--macro_param, page 446*.

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 337*.

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 68*.



On 64-bit architectures, the expression `#PC` becomes ambiguous in C-SPY macros. Therefore, when you debug a 64-bit MCU, you must use `#PC32` for AArch32 and `#PC64` for AArch64 in C-SPY macros. When debugging a 32-bit MCU, use `#PC`.

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:

`__message argList;`	Prints the output to the **Debug Log** window.
`__fmessage file, argList;`	Prints the output to the designated file.
`__smessage argList;`	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 371*.

To produce messages in the **Debug Log** window:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2, " in the Debug Log window.";
```

This produces this message in the **Debug Log** window:

```
This line prints the values 42 and 37 in the Debug Log window.
```

To write the output to a designated file:

```
__fmessage myfile, "Result is ", res, "!\n";
```

To produce strings:

```
myMacroVar = __smessage 42, " is the answer.";
```

`myMacroVar` now contains the string `"42 is the answer."`.

Specifying display format of arguments

To override the default display format of a scalar argument (number or pointer) in *`argList`*, suffix it with a `:` followed by a format specifier. Available specifiers are:

`%b`	for binary scalar arguments
`%o`	for octal scalar arguments
`%d`	for decimal scalar arguments
`%x`	for hexadecimal scalar arguments
`%c`	for character scalar arguments

These match the formats available in the **Watch** and **Locals** windows, but number prefixes and quotes around strings and characters are not printed.

Example:

```
__message "The character '", cvar:%c, "' has the decimal value ", cvar;
```

Depending on the value of the variables, this produces this message:

```
The character 'A' has the decimal value 65
```



A character enclosed in single quotes (a character literal) is an integer constant and is not automatically formatted as a character. For example:

```
__message 'A', " is the numeric value of the character ", 'A':%c;
```

would produce:

```
65 is the numeric value of the character A
```

Optionally, a number can be inserted between the `%` and the letter, to format an integer to that minimum width. Binary, octal, and hexadecimal numbers will be left-padded with zeros, decimal numbers and characters will be padded with spaces. Note that numbers that do not fit within the requested minimum width will *not* be truncated.

Examples:

```
__message 31:%4x;      // 001f
__message 31:%4d;      //   31
__message 31:%8b;      // 00011111
```



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 331*.

execConfigureTraceETM

Syntax

```
execConfigureTraceETM
```

For use with

All C-SPY hardware drivers where full instruction trace is supported and enabled.

Description

This macro is executed just before execution begins, and should be used for setting up device-specific registers that might be required to get full instruction trace (ETM/PTM) out to a physical pin, or to configure device-specific parts of an on-chip trace unit (ETB/MTB/PTB).

execConfigureTraceSWO

Syntax

```
execConfigureTraceSWO
```

For use with

All C-SPY hardware drivers where SWO trace is supported and enabled.

Description

This macro is executed just before execution begins, including when execution starts after being halted by a breakpoint and during single-stepping. Use it to set up device-specific registers that might be required to get SWO trace out to a physical pin.

Configuring SWO/ITM or TPIU should generally not be required here, because this is handled by the probe driver.

execUserAttach

Syntax

```
execUserAttach
```

For use with

All C-SPY hardware debugger drivers except CADI.

Description

Called after the debugger attaches to a running application at its current location without resetting the target system (the option **Attach to running target**).

Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.

execUserPreload

Syntax

```
execUserPreload
```

For use with

All C-SPY drivers.

Description

Called after communication with the target system is established and the target application is downloaded, but before the debug information that corresponds to the final application is read.

Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.



Do not use this macro if you are using a flash loader. Use the macro `execUserFlashInit` instead to perform early initializations required by the flash loader, see *execUserFlashInit, page 340*.

execUserExecutionStarted

Syntax

```
execUserExecutionStarted
```

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

```
execUserExecutionStopped
```

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

```
execUserFlashInit
```

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

```
execUserSetup
```

For use with

All C-SPY drivers.

Description

Called once after the target application is downloaded.

Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.



If you define interrupts or breakpoints in a macro file that is executed at system start (using `execUserSetup`) we strongly recommend that you also make sure that they are removed at system shutdown (using `execUserExit`). An example is available in `SetupSimple.mac`, see the tutorials in the Information Center.

The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time `execUserSetup` is executed again. This seriously affects the execution speed.

execUserFlashReset

Syntax

```
execUserFlashReset
```

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

```
execUserPreReset
```

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

```
execUserReset
```

For use with

All C-SPY drivers.

Description

Called each time just after the reset command is issued.

Implement this macro to set up and restore data.

execUserStatus

Syntax

```
execUserStatus()
```

Return value

Return value	Description
`-1`	The status is invalid. Ignore this status.
`0x20`	The CPU is not powered or clock gated. Do not access any core debug registers.

For use with

The C-SPY I-jet driver

The C-SPY CMSIS-DAP driver

Description

Use this macro to handle situations where core debug registers cannot be accessed by the debugger, for instance, when the core is in a deep sleep state or when the core power or clock is gated. Use a probe configuration file together with this macro to avoid accessing unpowered debug components.

The macro is called before the processor status is polled.

This macro overrides the `_ExecDeviceStatus` macro if that macro is present in the board configuration file.

execUserExit

Syntax

```
execUserExit
```

For use with

All C-SPY drivers.

Description

Called once when the debug session ends.

Implement this macro to save status data etc.

execUserFlashExit

Syntax

```
execUserFlashExit
```

For use with

The C-SPY hardware debugger drivers.

Description

Called once when the flash programming ends.

Implement this macro to save status data etc. This macro is useful for flash loader functionality.

execUserCoreConnect

Syntax

```
execUserCoreConnect
```

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

The C-SPY ST-LINK driver.

The C-SPY TI MSP-FET driver.

The C-SPY TI Stellaris driver.

The C-SPY TI XDS driver.

Description

Called immediately when connection with the probe is established.

Implement this macro to perform actions before connecting the CPU. This macro is useful for unlocking/erasing a secured device.

REFERENCE INFORMATION ON C-SPY SYSTEM MACROS

This section gives reference information about each of the C-SPY system macros.

Summary of C-SPY system macros

This table summarizes the pre-defined system macros:

Macro	Description
`__argCount`	Returns the number of arguments in a string. See the documentation in the *Flash Loader Development Guide*.
`__abortLaunch, page 347`	Aborts the launch of the debugger
`__bytes2Word16`	Extracts a 16-bit word from a buffer. See the documentation in the *Flash Loader Development Guide*.
`__bytes2Word32`	Extracts a 32-bit word from a buffer. See the documentation in the *Flash Loader Development Guide*.
`__cancelAllInterrupts, page 347`	Cancels all ordered interrupts
`__cancelInterrupt, page 347`	Cancels an interrupt
`__clearBreak, page 348`	Clears a breakpoint

Macro	Description
__closeFile, page 348	Closes a file that was opened by *__openFile, page 371*
__delay, page 349	Delays execution
__disableInterrupts, page 349	Disables generation of interrupts
__driverType, page 349	Verifies the driver type
__emulatorSpeed, page 350	Sets the emulator clock frequency
__emulatorStatusCheckOnRead, page 351	Enables or disables the verification of the CPSR register after each read operation
__enableInterrupts, page 351	Enables generation of interrupts
__evaluate, page 352	Interprets the input string as an expression and evaluates it
__expandVar, page 352	Expands all argument variables in a string
__fillMemory8, page 353	Fills a specified memory area with a byte value
__fillMemory16, page 354	Fills a specified memory area with a 2-byte value
__fillMemory32, page 354	Fills a specified memory area with a 4-byte value
__fillMemory64, page 355	Fills a specified memory area with an 8-byte value
__gdbserver_exec_command, page 356	Send strings or commands to the GDB Server
__getArg	Returns an argument from a string. See the documentation in the *Flash Loader Development Guide*.
__getNumberOfCores, page 356	Gets the number local cores being debugged.
__getSelectedCore, page 356	Gets the number of the current core.
__getTracePortSize, page 357	Returns the width of the trace port
__hasDAPRegs, page 358	Returns true if the C-SPY driver supports the macros __readAPReg, __readDPReg, and __writeAPReg, and __writeDPReg
__hwJetResetWithStrategy, page 358	Performs a hardware reset and a halt of the target CPU
__hwReset, page 359	Performs a hardware reset and a halt of the target CPU
__hwResetRunToBp, page 359	Performs a hardware reset and then executes to the specified address
__hwResetWithStrategy, page 360	Performs a hardware reset and halt with delay of the target CPU
__hwRunToBreakpoint, page 361	Sets a temporary breakpoint and starts the execution
__isBatchMode, page 362	Checks if C-SPY is running in batch mode or not.
__isMacroSymbolDefined, page 362	Checks if a C-SPY macro symbol is defined.
__jlinkExecCommand, page 363	Sends a low-level command to the J-Link/J-Trace driver
__jlinkExecMacro, page 363	Calls a function (without any parameter) in a J-Link script file.
__jtagCommand, page 364	Sends a low-level command to the JTAG instruction register
__jtagCP15IsPresent, page 364	Checks if coprocessor CP15 is available

Macro	Description
__jtagCP15ReadReg, page 365	Returns the coprocessor CP15 register value
__jtagCP15WriteReg, page 365	Writes to the coprocessor CP15 register
__jtagData, page 365	Sends a low-level data value to the JTAG data register
__jtagRawRead, page 366	Returns the read data from the JTAG interface
__jtagRawSync, page 366	Writes accumulated data to the JTAG interface
__jtagRawWrite, page 367	Accumulates data to be transferred to the JTAG
__jtagResetTRST, page 368	Resets the ARM TAP controller via the TRST JTAG signal
__loadImage, page 368	Loads a debug image
__makeString	Creates a new buffer string. See the documentation in the *Flash Loader Development Guide*.
__memoryRestore, page 369	Restores the contents of a file to a specified memory zone
__memorySave, page 370	Saves the contents of a specified memory area to a file
__messageBoxYesCancel, page 370	Displays a Yes/Cancel dialog box for user interaction
__messageBoxYesNo, page 371	Displays a Yes/No dialog box for user interaction
__openFile, page 371	Opens a file for I/O operations
__orderInterrupt, page 372	Generates an interrupt
__popSimulatorInterruptExecutingStack, page 373	Informs the interrupt simulation system that an interrupt handler has finished executing
__probeType, page 374	Verifies the probe type
__readAPReg, page 374	Reads from an AP register
__readDPReg, page 375	Reads from a DP register
__readFile, page 375	Reads from the specified file
__readFileByte, page 376	Reads one byte from the specified file
__readMemory8, __readMemoryByte, page 376	Reads one byte from the specified memory location
__readMemory16, page 377	Reads two bytes from the specified memory location
__readMemory32, page 377	Reads four bytes from the specified memory location
__readMemory64, page 377	Reads eight bytes from the specified memory location
__readMemoryBuffer	Reads bytes and returns them as a string. See the documentation in the *Flash Loader Development Guide*.
__registerMacroFile, page 378	Registers macros from the specified file
__resetFile, page 378	Rewinds a file opened by __openFile, page 371
__restoreSoftwareBreakpoints, page 379	Restores any breakpoints that were destroyed during system startup.
__selectCore, page 379	Switches focus from the current core to the specified core.

Macro	Description
__setCodeBreak, page 380	Sets a code breakpoint
__setDataBreak, page 381	Sets a data breakpoint
__setDataLogBreak, page 383	Sets a data log breakpoint
__setLogBreak, page 384	Sets a log breakpoint
__setSimBreak, page 385	Sets a simulation breakpoint
__setSystemMaxData, page 385	Sets the number of bytes to collect from an application started with `__system2` or `__system3`.
__setSystemTimeout, page 386	Sets the timeout for an application started with `__system1`, `__system2`, or `__system3`.
__setTraceStartBreak, page 387	Sets a trace start trigger breakpoint
__setTraceStopBreak, page 388	Sets a trace stop trigger breakpoint
__sourcePosition, page 389	Returns the file name and source location if the current execution location corresponds to a source location
__strFind, page 390	Searches a given string for the occurrence of another string
__subString, page 390	Extracts a substring from another string
__system1, page 391	Starts an external application
__system2, page 392	Starts an external application with `stdout` and `stderr` collected in one variable
__system3, page 392	Starts an external application with `stdout` and `stderr` collected in separate variables
__targetDebuggerVersion, page 393	Returns the version of the target debugger
__toLower, page 393	Returns a copy of the parameter string where all the characters have been converted to lower case
__toString, page 394	Prints strings
__toUpper, page 395	Returns a copy of the parameter string where all the characters have been converted to upper case
__unloadImage, page 395	Unloads a debug image
__wallTime_ms, page 396	Returns the current host computer CPU time in milliseconds
__whichCore, page 396	Returns the currently running core
__writeAPReg, page 397	Writes to an `AP` register
__writeDPReg, page 397	Writes to a `DP` register
__writeFile, page 398	Writes to the specified file
__writeFileByte, page 398	Writes one byte to the specified file
__writeMemory8, __writeMemoryByte, page 399	Writes one byte to the specified memory location
__writeMemory16, page 399	Writes a two-byte word to the specified memory location
__writeMemory32, page 399	Writes a four-byte word to the specified memory location
__writeMemory64, page 400	Writes an eight-byte word to the specified memory location

Macro	Description
`__writeMemoryBuffer`	Writes bytes from a buffer. See the documentation in the *Flash Loader Development Guide*.

Table 16. Summary of system macros

__abortLaunch

Syntax

```
__abortLaunch(message)
```

Parameters

param	A string.

Return value

None.

For use with

All C-SPY drivers.

Description

This macro can be used for ....

Example

```
if (!__messageBoxYesCancel("Do you want to mass erase to unlock the device?", "Unlocking device"))
{ __abortLaunch("Unlock canceled. Debug session cannot continue."); }
```

__cancelAllInterrupts

Syntax

```
__cancelAllInterrupts()
```

Return value

`int 0`

For use with

The C-SPY Simulator. (Not available for all cores and devices.)

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 17. __cancelInterrupt return values

For use with

The C-SPY Simulator. (Not available for all cores and devices.)

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 90.

__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.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

__disableInterrupts

Syntax

```
__disableInterrupts()
```

Return value

Result	Value
Successful	`int 0`
Unsuccessful	Non-zero error number

Table 18. __disableInterrupts return values

For use with

The C-SPY Simulator. (Not available for all cores and devices.)

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 `"cadi"` corresponds to the C-SPY CADI driver `"cmsisdap"` corresponds to the C-SPY CMSIS-DAP driver `"e2"` corresponds to the C-SPY E2 driver

"gdbserv" corresponds to the C-SPY GDB Server driver
"generic" corresponds to third-party drivers
"g+link pro" corresponds to the G+LINK driver
"ijet" corresponds to the C-SPY I-jet driver
"jlink" corresponds to the C-SPY J-Link/J-Trace driver
"lmiftdi" corresponds to the C-SPY TI Stellaris driver
"mspfet" corresponds to the C-SPY MSP-FET driver
"nulink" corresponds to the C-SPY Nu-Link driver
"xds" corresponds to the C-SPY TI XDS driver
"stlink" corresponds to the C-SPY ST-LINK driver

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 19. __driverType return values

For use with

All C-SPY drivers.

Description

Checks to see if the current C-SPY driver is identical to the driver type of the *driver_id* parameter.

Example

```
__driverType("sim")
```

If the simulator is the current driver, the value 1 is returned. Otherwise 0 is returned.

__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 20. __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.

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.



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.

For the C-SPY I-jet/JTAG-jet driver, this macro is recognized, but has no effect.

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 21. __enableInterrupts return values

For use with

The C-SPY Simulator. (Not available for all cores and devices.)

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 22. __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.

__expandVar

Syntax

```
__expandVar(argvar)
```

Parameters

argvar	A string that contains one or more predefined or custom argument variables for paths or arguments, for example an installation directory or an argument to a tool.

Return value

Result	Value
Successful	The string, with the argument variables expanded.
Unsuccessful	If any of the variables in the argument are not recognized, the return value is an empty string.

Table 23. __expandVar return values

For use with

All C-SPY drivers.

Description

Expands any Embedded Workbench argument variables—such as `$PROJ_DIR$` or `$PROJ_FNAME$`—contained in a string argument.

Example

```
__expandVar("$PROJ_DIR$")
```

returns, for example,

```
D:\Documents\myProject
```

and

```
__expandVar("$TOOLKIT_DIR$\\projects\\$PROJ_FNAME$.txt")
```

returns, for example,

```
D:\Applications\IAR\Embedded Workbench\projects\myproj.txt
```

See also

Information about argument variables in the *IDE Project Management and Building Guide for Arm*.

__fillMemory8

Syntax

```
__fillMemory8(value, address, zone, length, format)
```

Parameters

value	An integer that specifies the value.
address	An integer that specifies the memory start address.
zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.
length	An integer that specifies how many bytes are affected.
format	A string that specifies the exact fill operation to perform. Choose between:

`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.
`OR`	An `OR` 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.

Return value

`int 0`

For use with

All C-SPY drivers.

Description

Fills a specified memory area with a byte value.

Example

```
__fillMemory8(0x80, 0x700, "", 0x10, "OR");
```

__fillMemory16

Syntax

```
__fillMemory16(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 121*.
length	An integer that defines how many 2-byte entities to be affected.
format	A string that specifies the exact fill operation to perform. Choose between:

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.
OR	An OR 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.

Return value

int 0

For use with

All C-SPY drivers.

Description

Fills a specified memory area with a 2-byte value.

Example

```
__fillMemory16(0xCDCD, 0x7000, "", 0x200, "Copy");
```

__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 121*.
length	An integer that defines how many 4-byte entities to be affected.
format	A string that specifies the exact fill operation to perform. Choose between:

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.
OR	An OR 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.

Return value

int 0

For use with

All C-SPY drivers.

Description

Fills a specified memory area with a 4-byte value.

Example

```
__fillMemory32(0x0000FFFF, 0x4000, "", 0x1000, "XOR");
```

__fillMemory64

Syntax

```
__fillMemory64(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 121*.

length An integer that defines how many 8-byte entities to be affected.

format A string that specifies the exact fill operation to perform. Choose between:

Copy	*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.
OR	An OR 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.

Return value

int 0

For use with

All C-SPY drivers.

Description

Fills a specified memory area with an 8-byte value.

Example

```
__fillMemory64(0x0155'FFFF'FFFF'FFFF, 0x4000, "", 0x1000, "AND");
```

__gdbserver_exec_command

Syntax

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

Parameters

string String or command sent to the GDB Server. For more information, see the GDB server documentation.

For use with

The C-SPY GDB Server driver.

Description

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

__getNumberOfCores

Syntax

```
__getNumberOfCores()
```

Return value

The number of local cores being debugged.

For use with

The C-SPY simulator. (Not available for all cores and devices.)

The C-SPY I-jet driver.

Description

This macro returns the number of local cores being debugged.

Example

```
test ()
{
  __var i;
  for (i = 0; i < __getNumberOfCores(); i++ )
  {
    __selectCore(i);
    __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n";
  }
}
```

See also

__getSelectedCore, page 356 and *__selectCore, page 379*

__getSelectedCore

Syntax

```
__getSelectedCore()
```

Return value

The current core. The cores are numbered from `0` and upwards.

For use with

The C-SPY simulator. (Not available for all cores and devices.)

The C-SPY I-jet driver.

Description

Gets the number of the current core.

Example

```
test ()
{
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n";
  __selectCore(0);
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n";
  __selectCore(1);
  __message "Core: ", __getSelectedCore(), " pc = ", #PC:%x, "\n";
```

A typical result of the above macro would be (assuming that the original core was number `1`):

```
Core: 1 pc = 0000213C
Core: 0 pc = 00000494
Core: 1 pc = 0000213C
```

See also

__selectCore, page 379.

__getTracePortSize

Syntax

```
__getTracePortSize
```

Return value

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 driver.

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

Description

Returns the width of the trace port.

See also

ETM Trace Settings dialog box (I-jet), page 161 and *ETM Trace Settings dialog box (J-Link/J-Trace), page 163*, respectively.

__hasDAPRegs

Syntax

```
__hasDAPRegs()
```

Return value

Result	Value
The C-SPY driver supports the macros `__readAPReg`, `__readDPReg`, `__writeAPReg`, and `__writeDPReg` for the current CPU core.	`true`
The C-SPY driver does not support the macros `__readAPReg`, `__readDPReg`, `__writeAPReg`, and `__writeDPReg` 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

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 reset strategy number. For information about supported reset strategies, see *--jet_standard_reset, page 438*.

Return value

Result	Value
Successful. The delay feature is not supported by the debugging probe	`-1`
Unsuccessful. The reset strategy is not supported by the debugging probe	`-3`
Unsuccessful. Other	`-4`

Table 26. __hwJetResetWithStrategy return values

For use with

The C-SPY I-jet driver.

Description

Specifies the reset strategy to perform.

Example

```
__hwJetResetWithStrategy(0,2)
```

Performs a hardware reset.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

__hwReset

Syntax

```
__hwReset(halt_delay)
```

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.

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

```
__hwReset(0)
```

Resets the CPU and immediately halts it.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

__hwResetRunToBp

Syntax

```
__hwResetRunToBp(strategy, breakpoint_address, timeout)
```

Parameters

strategy	For information about supported reset strategies in the C-SPY I-jet driver, see *--jet_standard_reset, page 438*. For information about supported reset strategies in the C-SPY J-Link driver, see the J-Link/J-Trace documentation available at *www.segger.com* .
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.
-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 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.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

__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 driver only supports strategy 2 (hardware reset). For information about supported reset strategies in the C-SPY J-Link driver, see the J-Link/J-Trace documentation available at *www.segger.com*.

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

Result	Value
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 driver.

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

This macro also exists in other C-SPY hardware drivers, but 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.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

__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 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 milliseconds have passed.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

__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.

__isMacroSymbolDefined

Syntax

```
__isMacroSymbolDefined(symbol)
```

Parameters

symbol	The name of a C-SPY macro variable or macro function (a string).

Return value

`1` if *symbol* is an existing macro symbol. `0` if *symbol* is not defined.

For use with

All C-SPY drivers.

Description

This macro identifies whether a string is the name of an existing C-SPY macro symbol (variable or function) or not.

Example

```
__var someVariable;

...

if (__isMacroSymbolDefined("someVariable"))
  someVariable = 42;
else
  __message "The someVariable symbol is not defined!";
```

__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 J-Link/J-Trace documentation available at *www.segger.com*.

Example

For an example, see the J-Link/J-Trace documentation available at *www.segger.com*.

See also

--jlink_exec_command, page 442

__jlinkExecMacro

Syntax

```
__jlinkExecMacro(function)
```

Parameters

function	A string that defines a name of the function in a J-Link script.

Return value

An unsigned integer	The return value from the J-Link script function.

For use with

The C-SPY J-Link driver.

Description

Calls a function (without any parameters) in a J-Link script file.

Example

```
_jlinkExecMacro("myfunction");
```

See also

The J-Link/J-Trace documentation available at *www.segger.com*.

__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 driver.

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

Description

Checks if the coprocessor CP15 is available.

__jtagCP15ReadReg

Syntax

```
__jtagCP15ReadReg(CRn, CRm, op1, op2)
```

Parameters

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 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.

For use with

The C-SPY I-jet driver.

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

Description

Writes a value to the CP15 register.

__jtagData

Syntax

```
__jtagData(dr, bits)
```

Parameters

`dr`	A 32-bit data register value.
`bits`	The number of valid bits in `dr`, 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 `bits` 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

```
__jtagCommand(14);
Id = __jtagData(0,32);
```

Returns the JTAG ID of the Arm target device.

__jtagRawRead

Syntax

```
__jtagRawRead(bitpos, numbits)
```

Parameters

bitpos	The start bit position in the returned JTAG bits to return data from.
numbits	The number of bits to read. The maximum value is 32.

For use with

The C-SPY 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

tdi	The data output to the `TDI` pin. This data is sent with the least significant bit first.
tms	The data output to the `TMS` pin. This data is sent with the least significant bit first.
numbits	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 debug 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 debug image.
debugInfoOnly	A non-zero integer value if no code or data should be downloaded to the target system, which means that C-SPY will only read the debug information from the debug file. Or, 0 (zero) for download.

Return value

Value	Result
Non-zero integer number	A unique module identification.
`int 0`	Loading failed.

Table 33. __loadImage return values

For use with

All C-SPY drivers.

Description

Loads a debug image (debug file).



Images are only downloaded to RAM and no flash loading will be performed, unless you are using J-Link or TI MSP-FET.

Example

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

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 469 and *Loading multiple debug images, page 36*.

__memoryRestore

Syntax

```
__memoryRestore(zone, filename, offset)
```

Parameters

zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.
filename	A string that specifies the file to be read. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide for Arm*.
offset	An integer offset. When restoring data from the file into memory, this offset is added to the addresses specified in the file. For example, if the file contains data from `0x0–0x1FF` and the offset is `0x400`, the data will be placed in memory in the range `0x400–0x5FF`. This makes it possible to restore data into memory on addresses larger than 32-bit, even if the file format only supports 32-bit addresses.

Return value

`int 0`

For use with

All C-SPY drivers.

Description

Reads the contents of a file and saves it to the specified memory zone.

Example

```
__memoryRestore("", "c:\\temp\\saved_mem.hex", 0x400);
```

See also

Memory Restore dialog box, page 130.

__memorySave

Syntax

```
__memorySave(start, stop, format, filename, zerostart)
```

Parameters

start	A string that specifies the first location of the memory area to be saved.
stop	A string that specifies the last location of the memory area to be saved.
format	A string that specifies the format to be used for the saved memory. Choose between: `intel-extended` `motorola` `motorola-s19` `motorola-s28` `motorola-s37`
filename	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 *IDE Project Management and Building Guide for Arm*.
zerostart	An integer. If it is `1` (or any non-zero value), the addresses in the saved file will start from `0x0`. For example, if the specified memory range is `0x400–0x5FF`, the address range in the file will be `0x0–0x1FF`. This makes it possible to save memory from addresses larger than 32-bit to file formats which only support 32-bit addresses. If the parameter is `0`, the file will contain the specified addresses as given.

Return value

`int 0`

For use with

All C-SPY drivers.

Description

Saves the contents of a specified memory area to a file.

Example

```
__memorySave(":0x00", ":0xFF", "intel-extended", "c:\\temp\\saved_memory.hex", 0);
```

See also

Memory Save dialog box, page 129.

__messageBoxYesCancel

Syntax

```
__messageBoxYesCancel(message, caption)
```

Parameters

message	A message that will appear in the message box.
caption	The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 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(message, caption)
```

Parameters

message	A message that will appear in the message box.
caption	The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 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
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 `__openFile` can specify a location relative to this directory. In addition, you can use argument variables such as `$PROJ_DIR$` and `$TOOLKIT_DIR$` in the path argument.

Example

```
__var myFileHandle;         /* The macro variable to contain */
                            /* the file handle */
myFileHandle = __openFile("$PROJ_DIR$\\Debug\\Exe\\test.tst", "r");
if (myFileHandle)
{
    /* successful opening */
}
```

See also

For information about argument variables, see the *IDE Project Management and Building Guide for Arm*.

__orderInterrupt

Syntax

```
__orderInterrupt(specification, first_activation, repeat_interval, variance,
                 infinite_hold_time, hold_time, probability)
```

Parameters

specification The interrupt (string). The specification can either be the full specification used in the device description file (`ddf`) or only the name. In the latter case the interrupt system will automatically get the description from the device description file.

first_activation	The first activation time in cycles (integer)
repeat_interval	The periodicity in cycles (integer)
variance	The timing variation range in percent (integer between 0 and 100)
infinite_hold_time	1 if infinite, otherwise 0.
hold_time	The hold time (integer)
probability	The probability in percent (integer between 0 and 100)

Return value

The macro returns an interrupt identifier (`unsigned long`).

If the syntax of *specification* is incorrect, it returns -1.

For use with

The C-SPY simulator. (Not available for all cores and devices.)

Description

Generates an interrupt.

Example

This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles:

```
__orderInterrupt( "IRQ", 4000, 2000, 0, 1, 0, 100 );
```

__popSimulatorInterruptExecutingStack

Syntax

```
__popSimulatorInterruptExecutingStack(void)
```

Return value

`int 0`

For use with

The C-SPY simulator. (Not available for all cores and devices.)

Description

Informs the interrupt simulation system that an interrupt handler has finished executing, as if the normal instruction used for returning from an interrupt handler was executed.

This is useful if you are using interrupts in such a way that the normal instruction for returning from an interrupt handler is not used, for example in an operating system with task-switching. In this case, the interrupt simulation system cannot automatically detect that the interrupt has finished executing.

See also

Simulating an interrupt in a multi-task system, page 311.

__probeType

Syntax

```
__probeType(probe_id)
```

Parameters

probe_id A string corresponding to the probe you want to check for. Choose one of these:
`"I-jet"` corresponds to the I-jet probe
`"I-jet-Trace"` corresponds to the I-jet Trace probe.

The strings are case-insensitive.

Return value

Result	Value
Successful	`1`
Unsuccessful	`0`

Table 37. __probeType return values

For use with

The C-SPY I-jet driver.

Description

Checks to see if the current connected probe is identical to the probe type of the *probe_id* parameter.

Example

```
__probeType("I-jet")
```

If I-jet is the current connected probe, the value `1` is returned. Otherwise `0` is returned.

__readAPReg

Syntax

```
__readAPReg(register)
```

Parameters

register An 8-bit `AP` register offset.

Return value

Result	Value
Successful	`true`
Unsuccessful	`false`

Table 38. __readAPReg return values

For use with

The C-SPY I-jet 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.

__readDPReg

Syntax

```
__readDPReg(register)
```

Parameters

register	An 8-bit DP register offset.

Return value

Result	Value
Successful	true
Unsuccessful	false

Table 39. __readDPReg return values

For use with

The C-SPY I-jet 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 40. __ readFile return values

For use with

All C-SPY drivers.

Description

Reads a sequence of hexadecimal digits from the given file and converts them to an unsigned long which is assigned to the *value* parameter, which should be a pointer to a macro variable.

Only printable characters representing hexadecimal digits and white-space characters are accepted, no other characters are allowed.

Example

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

In this example, if the file pointed to by `myFileHandle` contains the ASCII characters `1234 abcd 90ef`, consecutive reads will assign the values `0x1234 0xabcd 0x90ef` to the variable `number`.

__readFileByte

Syntax

```
__readFileByte(fileHandle)
```

Parameters

fileHandle	A macro variable used as filehandle by the `__openFile` macro.

Return value

`-1` upon error or end-of-file, otherwise a value between `0` and `255`.

For use with

All C-SPY drivers.

Description

Reads one byte from a file.

Example

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

__readMemory8, __readMemoryByte

Syntax

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

Parameters

address	The memory address (integer).
zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.

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

```
__readMemory8(0x0108, "");
```

__readMemory16

Syntax

```
__readMemory16(address, zone)
```

Parameters

address	The memory address (integer).
zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.

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 121*.

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, "");
```

__readMemory64

Syntax

```
__readMemory64(address, zone)
```

Parameters

address	The memory address (integer).
zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.

Return value

The macro returns the value from memory.

For use with

All C-SPY drivers.

Description

Reads an eight-byte word from a given memory location.

Example

```
__readMemory64(0x8000, "");
```

__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 332.

__resetFile

Syntax

```
__resetFile(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

Rewinds a file previously opened by __openFile.

__restoreSoftwareBreakpoints

Syntax

```
__restoreSoftwareBreakpoints()
```

Return value

int 0

For use with

All C-SPY hardware drivers.

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 initialize by copy directive for code in the linker configuration file or if you have any __ramfunc 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

```
__selectCore(int core)
```

Parameters

core The core to switch to. The cores are numbered from 0 and upwards.

Return value

int 0

For use with

The C-SPY simulator. (Not available for all cores and devices.)

The C-SPY I-jet driver.

Description

Switches focus from the current core to the specified core for the duration of the macro invocation or until any next invocation of __selectCore.

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

__getSelectedCore, page 356.

__setCodeBreak

Syntax

```
__setCodeBreak(location, count, condition, cond_type, action)
```

Parameters

location	A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box, page 117*.
count	An integer that specifies the number of times that a breakpoint condition must be fulfilled before a break occurs the next time.
condition	The breakpoint condition. This must be a valid C-SPY expression, for instance a C-SPY macro function.
cond_type	The condition type; either `"CHANGED"` or `"TRUE"` (string).
action	An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 41. __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.

Example

```
__setCodeBreak("{D:\\src\\prog.c}.12.9",
3, "d>16", "TRUE", "ActionCode()");
```

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

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

This example sets the breakpoint within a specific source file and line without using the absolute file path to the source:

```
__setCodeBreak("{main.c}.288.7",
0, "1", "TRUE", "");
```

See also

Breakpoints, page 90.

__setDataBreak

Syntax

In the simulator:

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

In the C-SPY I-jet driver and in the C-SPY CMSIS-DAP 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 information about the location types, see *Enter Location dialog box, page 117*.
count	An integer that specifies the number of times that a breakpoint condition must be fulfilled before a break occurs the next time. This parameter only applies to the simulator.
condition	The breakpoint condition (string). This parameter only applies to the simulator.
cond_type	The condition type; either `"CHANGED"` or `"TRUE"` (string). This parameter only applies to the simulator.
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 only applies to the simulator.
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 only applies to the C-SPY I-jet driver and the C-SPY CMSIS-DAP driver.

match	Enables matching of the accessed data. Choose between "TRUE"or "FALSE". This parameter only applies to the C-SPY I-jet driver and the C-SPY CMSIS-DAP driver.
data	A data value to match, in unsigned 32-bit format. This parameter only applies to the C-SPY I-jet driver and the C-SPY CMSIS-DAP driver.
mask	Specifies which part of the data value to match—word, halfword, or byte—in unsigned 32-bit format. This parameter only applies to the C-SPY I-jet driver and the C-SPY CMSIS-DAP driver

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. __setDataBreak return values

For use with

The C-SPY simulator.

The C-SPY CMSIS-DAP driver.

The C-SPY GDB Server driver.

The C-SPY G+LINK driver.

The C-SPY I-jet driver.

The C-SPY J-Link driver.

The C-SPY PE micro driver.

The C-SPY ST-Link driver.

The C-SPY TI MSP-FET driver.

The C-SPY TI Stellaris driver.

The C-SPY TI XDS 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);
```

For I-jet:

```
__var brk;
brk = __setDataBreak("myVar", "W", "FALSE", "TRUE",
      0xABCD, 0xFFFF);
...
__clearBreak(brk);
```

See also

Breakpoints, page 90.

__setDataLogBreak

Syntax

```
__setDataLogBreak(variable, access, extend)
```

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.
extend	Use extended range: `"TRUE"` or `"FALSE"`. This parameter is only available when using a device and a debug probe that support SWO.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 43. __setDataLogBreak return values

For use with

The C-SPY simulator. (Not available for all cores and devices.)

A device and a debug probe with support for SWO.

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

For the simulator:

```
__var brk;
brk = __setDataLogBreak("MyVar",
"R");
...
__clearBreak(brk);
```

For the C-SPY I-jet driver:

```
__var brk;
brk = __seDataLogBreak("myVar", "RW", "FALSE");
...
__clearBreak(brk);
```

See also

Breakpoints, page 90 and *Getting started using data logging, page 198*.

__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 117*.
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 44. __setLogBreak return values

For use with

All C-SPY drivers.

Description

Sets a log breakpoint, that is, a breakpoint which is triggered when an instruction is fetched from the specified location. If you have set the breakpoint on a specific machine instruction, the breakpoint will be triggered and the execution will temporarily halt and print the specified message in the C-SPY **Debug Log** window.

Example

```
__var logBp1;
__var logBp2;

logOn()
{
  logBp1 = __setLogBreak ("{C:\\temp\\Utilities.c}.23.1",
  "\"Entering trace zone at
:\", #PC:%X", "ARGS", "1", "TRUE");
  logBp2 = __setLogBreak ("{C:\\temp\\Utilities.c}.30.1",
    "Leaving trace zone...", "TEXT", "1", "TRUE");
}
```

```
logOff()
{
  __clearBreak(logBp1);
  __clearBreak(logBp2);
}
```

See also

Formatted output, page 337 and *Breakpoints, page 90*.

__setSimBreak

Syntax

```
__setSimBreak(location, access, action)
```

Parameters

location	A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For information about the location types, see *Enter Location dialog box, page 117*.
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 45. __setSimBreak return values

For use with

The C-SPY simulator. (Not available for all cores and devices.)

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.

__setSystemMaxData

Syntax

```
__setSystemMaxData(size)
```

Parameters

size	The maximum number of bytes to collect.

Return value

The most recent value set by this macro, or `65535` (the default value) if the macro has not been called before.

For use with

All C-SPY drivers.

Description

Use this macro to set the maximum number of bytes that can be collected from an external application started with the `__system2` or `__system3` macro. By default, a maximum of 65,535 bytes can be collected.

Example

```
__var exitCode;
__var out;
__var oldSize;

oldSize = __setSystemMaxData(100000); // Increasing the maximum data size

exitCode = __system2("dir /S", &out);

__message "Previous data limit:";
__message oldSize

__message "Output from the dir command:";
__message out;
```

See also

__setSystemTimeout, page 386, *__system2, page 392*, and *__system3, page 392*.

__setSystemTimeout

Syntax

```
__setSystemTimeout(timeout)
```

Parameters

timeout	The timeout in milliseconds.

Return value

The most recent value set by this macro, or `10000` (the default value) if the macro has not been called before.

For use with

All C-SPY drivers.

Description

Use this macro to set the timeout for an external application started with one of the macros `__system1`, `__system2`, or `__system3`. By default, the timeout is set to 10,000 milliseconds (10 seconds).

Example

```
__var exitCode;
__var out;
__var oldSize;
__var oldTimeout;
```

```
oldSize = __setSystemMaxData(100000);
oldTimeout = __setSystemTimeout(60000); // Changing the timeout to one minute

exitCode = __system2("dir /S", &out);

__message "Previous data limit:";
__message oldSize

__message "Previous timeout:";
__message oldTimeout;

__message "Output from the dir command:";
__message out;
```

See also

__setSystemMaxData, page 385, *__system1, page 391*, *__system2, page 392*, and *__system3, page 392*.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

__setTraceStartBreak

Syntax

In the simulator:

```
__setTraceStartBreak(location)
```

In the I-jet 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 117*.
access	The memory access type—`"F"` for fetch, `"R"` for read, `"W"` for write, or `"RW"` for read/write. This parameter only applies to I-jet.
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 only applies to I-jet.
match	Enables matching of the accessed data. Choose between `"TRUE"`or `"FALSE"`. This parameter only applies to I-jet.
data	A data value to match, in unsigned 32-bit format. This parameter only applies to I-jet.
mask	Specifies which part of the data value to match (word, halfword, or byte), in unsigned 32-bit format. This parameter only applies to I-jet.

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 46. __setTraceStartBreak return values

For use with

The C-SPY simulator. (Not available for all cores and devices.)

The C-SPY I-jet driver.

Description

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

Example

```
__var startTraceBp;
__var stopTraceBp;

traceOn()
{
  startTraceBp = __setTraceStartBreak
    ("{C:\\TEMP\\Utilities.c}.23.1");
  stopTraceBp = __setTraceStopBreak
    ("{C:\\temp\\Utilities.c}.30.1");
}

traceOff()
{
  __clearBreak(startTraceBp);
  __clearBreak(stopTraceBp);
}
```

See also

Trace Start Trigger breakpoint dialog box, page 179.

__setTraceStopBreak

Syntax

In the simulator:

```
__setTraceStopBreak(location)
```

In the I-jet driver:

```
__setTraceStopBreak(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 117*.

access — The memory access type: "F" for fetch, "R" for read, "W" for write, or "RW" for read/write.

This parameter only applies to I-jet.

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"` and `"FALSE"`.

This parameter only applies to I-jet.

match Enables matching of the accessed data. Choose between `"TRUE"`or `"FALSE"`.

This parameter only applies to I-jet.

data A data value to match, in unsigned 32-bit format.

This parameter only applies to I-jet.

mask Specifies which part of the data value to match (word, halfword, or byte), in unsigned 32-bit format.

This parameter only applies to I-jet.

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 47. __setTraceStopBreak return values

For use with

The C-SPY simulator. (Not available for all cores and devices.)

The C-SPY I-jet driver.

Description

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

Example

See *__setTraceStartBreak, page 387*.

See also

Trace Stop Trigger breakpoint dialog box, page 180.

__sourcePosition

Syntax

```
__sourcePosition(linePtr, colPtr)
```

Parameters

linePtr Pointer to the variable storing the line number

colPtr Pointer to the variable storing the column number

Return value

Result	Value
Successful	Filename string

Result	Value
Unsuccessful	Empty ("") string

Table 48. __sourcePosition return values

For use with

All C-SPY drivers.

Description

If the current execution location corresponds to a source location, this macro returns the filename as a string. It also sets the value of the variables, pointed to by the parameters, to the line and column numbers of the source location.

__strFind

Syntax

```
__strFind(macroString, pattern, position)
```

Parameters

macroString	A macro string.
pattern	The string pattern to search for
position	The position where to start the search. The first position is `0`

Return value

The position where the pattern was found or `-1` if the string is not found.

For use with

All C-SPY drivers.

Description

This macro searches a given string (*macroString*) for the occurrence of another string (*pattern*).

Example

```
__strFind("Compiler", "pile", 0)  = 3
__strFind("Compiler", "foo", 0)   = -1
```

See also

Macro strings, page 336.

__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 336.

__system1

Syntax

```
__system1(string)
```

Parameters

string — The command line used to start an external application. In some cases, the full path is needed. If it contains space characters, quotation marks escaped with backslashes (`\"`) can be added to encapsulate the path, and, separately, the arguments to the application, like this: `"\"D:\\My projects\\my app\\app.exe\" \"some argument\""`.

Return value

The exit code returned from the external application. If the application could not be launched or fails to return an appropriate exit code, `1` is returned.

For use with

All C-SPY drivers.

Description

This macro launches an external application. It ignores all output returned from the application, and terminates the launched application if the application has not finished within the current timeout setting (by default 10 seconds).

Example

```
__var exitCode;

exitCode = __system1("mkdir tmp");
```

See also

__setSystemTimeout, page 386, *__system2, page 392*, and *__system3, page 392*.

__system2

Syntax

```
__system2(string, &output)
```

Parameters

string	The command line used to start an external application. In some cases, the full path is needed. If it contains space characters, quotation marks escaped with backslashes (\") can be added to encapsulate the path, and, separately, the arguments to the application, like this: `"\"D:\\My projects\\my app\\app.exe\" \"some argument\""`.
output	The output returned from the application. Both the `stdout` and the `stderr` streams are stored in this variable.

Return value

The exit code returned from the external application. If the application could not be launched or fails to return an appropriate exit code, `1` is returned.

For use with

All C-SPY drivers.

Description

This macro launches an external application. The output from both the `stdout` and the `stderr` streams is stored in `output`. If no data has been received from the launched application within the timeout setting, or when the returned data exceeds the maximum data setting, the application is terminated. This restriction prevents the Embedded Workbench IDE from freezing or crashing because of misbehaving applications. By default, the timeout setting is 10 seconds, and the maximum data setting is 65,535 bytes.

Example

```
__var exitCode;
__var out_err;

exitCode = __system2("dir /S", &out_err);

__message "Output from the dir command:";
__message out_err;
```

__setSystemTimeout, page 386, *__setSystemMaxData, page 385*, *__system1, page 391*, and *__system3, page 392*.

__system3

Syntax

```
__system3(string, &output, &error)
```

Parameters

string	The command line used to start an external application. In some cases, the full path is needed. If it contains space characters, quotation marks escaped with backslashes (\") can be added to encapsulate the path, and, separately, the arguments to the application, like this: `"\"D:\\My projects\\my app\\app.exe\" \"some argument\""`.

output	The output returned from the `stdout` output stream of the application.
error	The output returned from the `stderr` output stream of the application.

Return value

The exit code returned from the external application. If the application could not be launched or fails to return an appropriate exit code, `1` is returned.

For use with

All C-SPY drivers.

Description

This macro launches an external application. The output from the `stdout` stream is stored in `output` and the `stderr` stream is stored in `error`. If no data has been received from the launched application within the timeout setting, or when the returned data exceeds the maximum data setting, the application is terminated. This restriction prevents the Embedded Workbench IDE from freezing or crashing because of misbehaving applications. By default, the timeout setting is 10 seconds, and the maximum data setting is 65,535 bytes.

Example

```
__var exitCode;
__var out;
__var err;

exitCode = __system3("dir /S", &out, &err);

__message "Output from the dir command:";
__message out;

__message "Error text from the dir command:";
__message err;
```

__setSystemTimeout, page 386, *__setSystemMaxData, page 385*, *__system1, page 391*, and *__system2, page 392*.

__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 336.

__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 336.

__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 336.

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

Table 49. __unloadImage return values

For use with

All C-SPY drivers.

Description

Unloads debug information from an already downloaded debug image.

See also

Loading multiple debug images, page 36 and *Images, page 469*.

__wallTime_ms

Syntax

```
__wallTime_ms()
```

Return value

Returns the current host computer CPU time in milliseconds.

For use with

All C-SPY drivers.

Description

This macro returns the current host computer CPU time in milliseconds. The first call will always return `0`.

Example

```
__var t1;
__var t2;

t1 = __wallTime_ms();
__var i;
for (i =0; i < 1000; i++)
  __message "Tick";
t2 = __wallTime_ms();
  __message "Elapsed time: ", t2 - t1;
```

__whichCore

Syntax

```
__whichCore()
```

Return value

Returns the currently running core. The cores are numbered from `0` and upwards.

For use with

All C-SPY drivers.

Description

When used in target execution, for example in a breakpoint condition or action, this macro returns the number of the core currently executing the application. Note that using this macro outside target execution will return `-1`.

Example

```
logAndMaybeStop()
{
    __var core;
    core = __whichCore();
    __message "Breakpoint hit by core ", core;
return core == 2;
}
```

When used as a breakpoint condition, the function will log every hit but stop only when it hits core 2.

__writeAPReg

Syntax

```
__writeAPReg(data, register)
```

Parameters

data	A 32-bit value.
register	An 8-bit AP register offset.

Return value

Result	Value
Successful	true
Unsuccessful	false

Table 50. __writeAPReg return values

For use with

The C-SPY I-Jet 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.

__writeDPReg

Syntax

```
__writeDPReg(data, register)
```

Parameters

data	A 32-bit value.
register	An 8-bit DP register offset.

Return value

Result	Value
Successful	true
Unsuccessful	false

Table 51. __writeDPReg return values

For use with

The C-SPY I-Jet 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*.



The __fmessage statement can do the same thing. The __writeFile macro is provided for symmetry with __readFile.

__writeFileByte

Syntax

```
__writeFileByte(fileHandle, value)
```

Parameters

fileHandle	A macro variable used as filehandle by the __openFile macro.
value	An integer.

Return value

int 0

For use with

All C-SPY drivers.

Description

Writes one byte to the file *fileHandle*.

__writeMemory8, __writeMemoryByte

Syntax

```
__writeMemory8(value, address, zone)
__writeMemoryByte(value, address, zone)
```

Parameters

value	An integer.
address	The memory address (integer).
zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.

Return value

`int 0`

For use with

All C-SPY drivers.

Description

Writes one byte to a given memory location.

Example

```
__writeMemory8(0x2F, 0x8020, "");
```

__writeMemory16

Syntax

```
__writeMemory16(value, address, zone)
```

Parameters

value	An integer.
address	The memory address (integer).
zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.

Return value

`int 0`

For use with

All C-SPY drivers.

Description

Writes two bytes to a given memory location.

Example

```
__writeMemory16(0x2FFF, 0x8020, "");
```

__writeMemory32

Syntax

```
__writeMemory32(value, address, zone)
```

Parameters

value	An integer.
address	The memory address (integer).
zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.

Return value

`int 0`

For use with

All C-SPY drivers.

Description

Writes four bytes to a given memory location.

Example

```
__writeMemory32(0x5555FFFF, 0x8020, "");
```

__writeMemory64

Syntax

```
__writeMemory64(value, address,
zone)
```

Parameters

value	An integer.
address	The memory address (integer).
zone	A string that specifies the memory zone, see *C-SPY memory zones, page 121*.

Return value

`int 0`

For use with

All C-SPY drivers.

Description

Writes eight bytes to a given memory location.

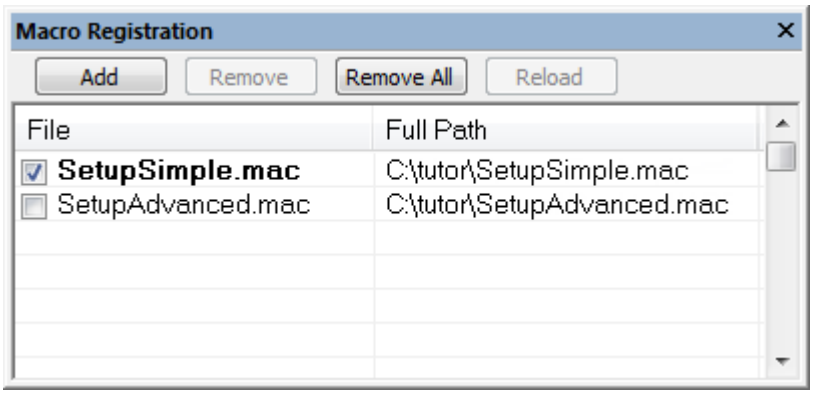
Example

```
__writeMemory64(0xFFFF'FFFF'8000'0000, 0xFFFF'8000, "");
```

GRAPHICAL ENVIRONMENT FOR MACROS

Macro Registration window

The **Macro Registration** window is available from the **View>Macros** submenu during a debug session.



Use this window to list, register, and edit your debugger macro files.

Double-click a macro file to open it in the editor window and edit it.

See also *Registering C-SPY macros—an overview, page 332*.

Requirements

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

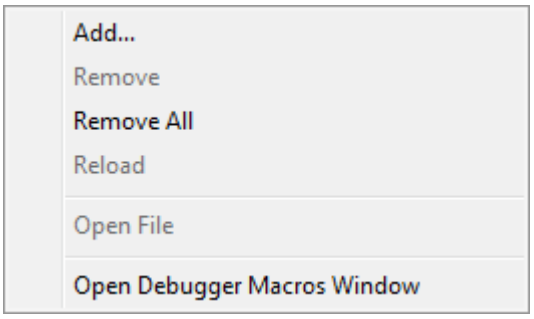
Display area

This area contains these columns:

File	The name of an available macro file. To register the macro file, select the check box to the left of the filename. The name of a registered macro file appears in bold style.
Full path	The path to the location of the added macro file.

Context menu

This context menu is available:

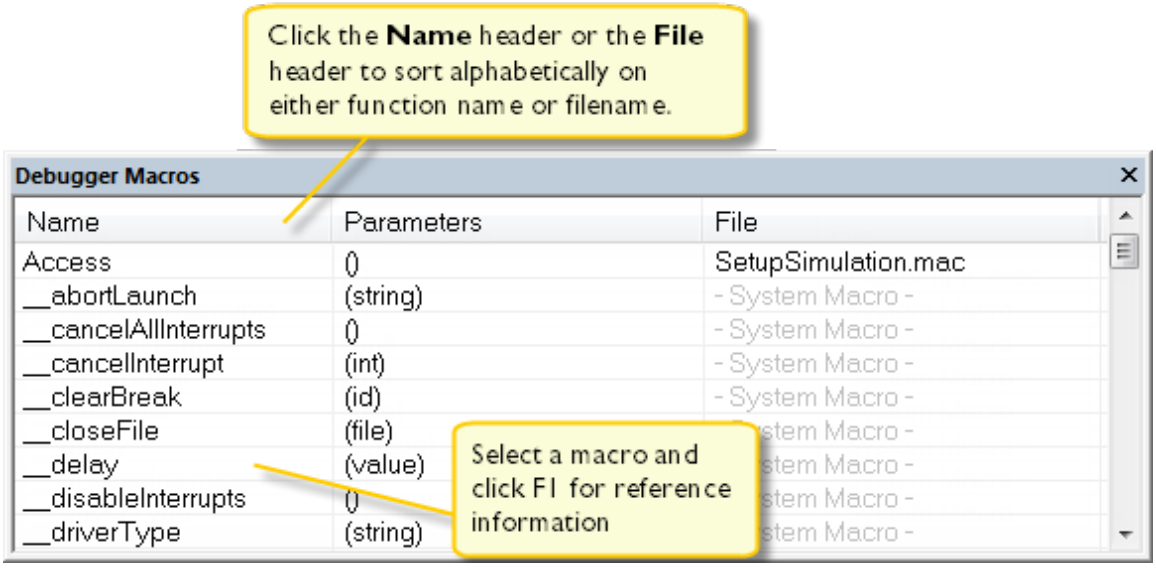


These commands are available:

Add	Opens a file browser where you can locate the macro file that you want to add to the list. This menu command is also available as a function button at the top of the window.
Remove	Removes the selected debugger macro file from the list. This menu command is also available as a function button at the top of the window.
Remove All	Removes all macro files from the list. This menu command is also available as a function button at the top of the window.
Reload	Registers the selected macro file. Typically, this is useful when you have edited a macro file. This menu command is also available as a function button at the top of the window.
Open File	Opens the selected macro file in the editor window.
Open Debugger Macros Window	Opens the **Debugger Macros** window.

Debugger Macros window

The **Debugger Macros** window is available from the **View>Macros** submenu during a debug session.



Use this window to list all registered debugger macro functions, either predefined system macros or your own. This window is useful when you edit your own macro functions and want an overview of all available macros that you can use.

- Click the column headers **Name** or **File** to sort alphabetically on either function name or filename.
- Double-clicking a macro defined in a file opens that file in the editor window.
- To open a macro in the **Macro Quicklaunch** window, drag it from the **Debugger Macros** window and drop it in the **Macro Quicklaunch** window.

Requirements

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

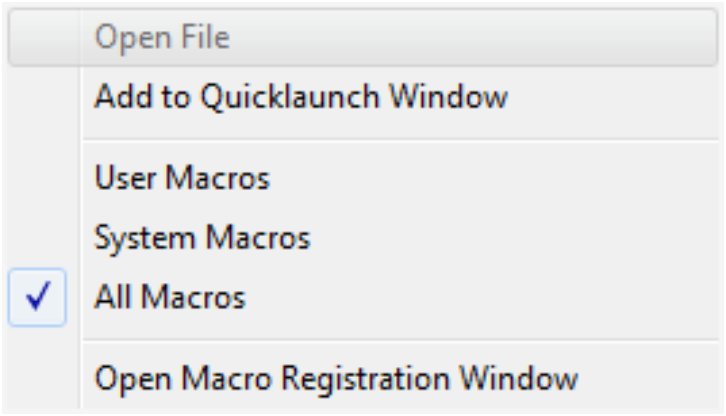
Display area

This area contains these columns:

Name	The name of the debugger macro.
Parameters	The parameters of the debugger macro.
File	For macros defined in a file, the name of the file is displayed. For predefined system macros, `-System Macro-` is displayed.

Context menu

This context menu is available:

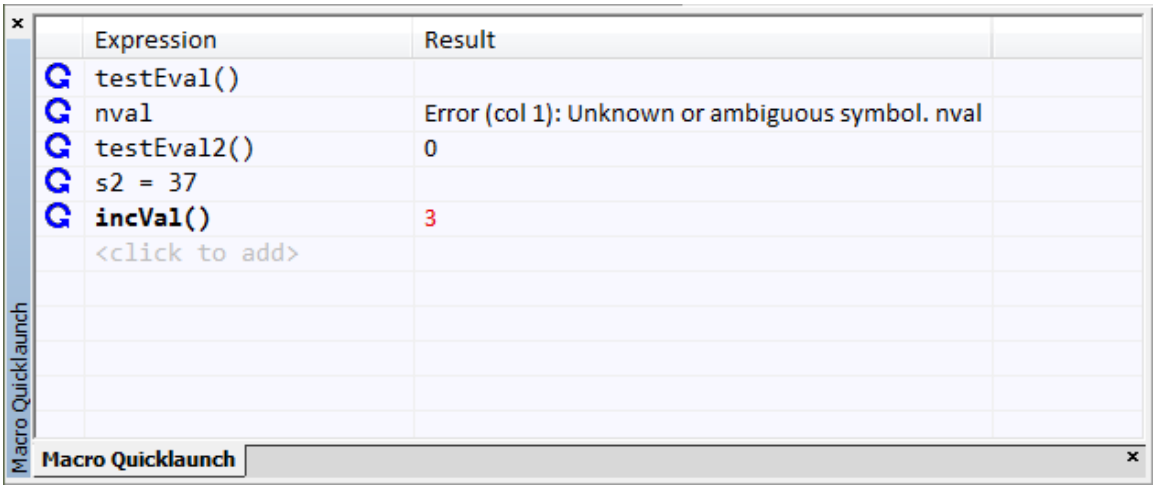


These commands are available:

Open File	Opens the selected debugger macro file in the editor window.
Add to Quicklaunch Window	Adds the selected macro to the **Macro Quicklaunch** window.
User Macros	Lists only the debugger macros that you have defined yourself.
System Macros	Lists only the predefined system macros.
All Macros	Lists all debugger macros, both predefined system macros and your own.
Open Macro Registration Window	Opens the **Macro Registration** window.

Macro Quicklaunch window

The **Macro Quicklaunch** window is available from the **View** menu.



Use this window to evaluate expressions, typically C-SPY macros.

For some devices, there are predefined C-SPY macros available with device support, typically provided by the chip manufacturer. These macros are useful for performing certain device-specific tasks. The macros are available in the **Macro Quicklaunch** window and are easily identified by their green icon.

The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is primarily designed for evaluating C-SPY macros. The window gives you precise control over when to evaluate an expression.

See also *Executing C-SPY macros—an overview, page 332*.

To add an expression:

Choose one of these alternatives:

- Drag the expression to the window
- In the **Expression** column, type the expression you want to examine.

If the expression you add and want to evaluate is a C-SPY macro, the macro must first be registered, see *Registering C-SPY macros—an overview, page 332*.

To evaluate an expression:

 Double-click the **Recalculate** icon to calculate the value of that expression.

Requirements

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

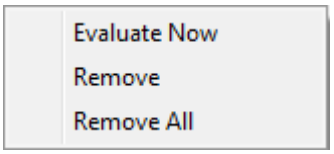
Display area

This area contains these columns:

Recalculate icon	 To evaluate the expression, double-click the icon. The latest evaluated expression appears in bold style.
Expression	One or several expressions that you want to evaluate. Click `<click to add>` to add an expression. If the return value has changed since last time, the value will be displayed in red.
Result	Shows the return value from the expression evaluation.

Context menu

This context menu is available:



These commands are available:

Evaluate Now	Evaluates the selected expression.
Remove	Removes the selected expression.
Remove All	Removes all selected expressions.

The C-SPY command line utility—cspybat

The C-SPY command line utility—cspybat

Contents

Using C-SPY in batch mode ... 409
Starting cspybat ... 409
Output ... 409
Invocation syntax ... 409
Summary of C-SPY command line options ... 410
General cspybat options ... 410
Options available for all C-SPY drivers ... 411
Options available for the simulator driver ... 411
Options available for the C-SPY GDB Server driver ... 412
Options available for the C-SPY I-jet driver ... 412
Options available for the C-SPY CMSIS-DAP driver ... 413
Options available for the C-SPY J-Link/J-Trace driver ... 414
Options available for the C-SPY TI MSP-FET driver ... 414
Options available for the C-SPY TI Stellaris driver ... 415
Options available for the C-SPY TI XDS driver ... 415
Options available for the C-SPY ST-LINK driver ... 415
Options available for the C-SPY third-party drivers ... 416
Reference information on C-SPY command line options ... 416
--application_args ... 416
--attach_to_running_target ... 417
--backend ... 417
--board_file ... 417
--code_coverage_file ... 418
--cycles ... 418
--debug_auth_enforce ... 419
--debug_auth_settings ... 419
--debug_auth_type ... 420
--debug_file ... 420
--device ... 420
--device_macro ... 421
--disable_interrupts ... 421
--download_only ... 422
--drv_catch_exceptions ... 422
--drv_communication ... 423
--drv_debugger_cache ... 425
--drv_default_breakpoint ... 426
--drv_enforce_mem_config ... 426
--drv_exclude_from_verify ... 427
--drv_interface ... 427
--drv_interface_speed ... 428
--drv_mem_ap ... 429
--drv_reset_to_cpu_start ... 429
--drv_restore_breakpoints ... 430
--drv_swo_clock_setup ... 430
--drv_trace_settings ... 431
--drv_vector_table_base ... 432
-f ... 432

--flash_loader ... 433
--function_profiling ... 433
--gdbserv_exec_command ... 434
--jet_board_cfg ... 434
--jet_board_did ... 434
--jet_cpu_clock ... 435
--jet_disable_pmu ... 436
--jet_disable_pmu_dap ... 436
--jet_ir_length ... 436
--jet_power_from_probe ... 437
--jet_probe ... 437
--jet_script_file ... 437
--jet_standard_reset ... 438
--jet_startup_connection_timeout ... 439
--jet_swo_on_d0 ... 439
--jet_swo_prescaler ... 440
--jet_swo_protocol ... 440
--jet_tap_position ... 441
--jlink_dcc_timeout ... 441
--jlink_device_select ... 441
--jlink_exec_command ... 442
--jlink_initial_speed ... 442
--jlink_ir_length ... 443
--jlink_reset_strategy ... 443
--jlink_script_file ... 444
--jlink_trace_source ... 444
--leave_target_running ... 445
--lmiftdi_reset_strategy ... 445
--macro ... 445
--macro_param ... 446
--mapu ... 446
--mspfet_erase_flash ... 447
--mspfet_interface_speed ... 447
--mspfet_reset_strategy ... 448
--mspfet_settlingtime ... 448
--mspfet_vccvoltage ... 448
--multicore_nr_of_cores ... 449
-p ... 449
--plugin ... 450
--proc_stack_*stack* ... 450
--reset_style ... 451
--sdm_debug_architecture ... 452
--sdm_library ... 452
--sdm_library_hint ... 453
--sdm_manifest ... 453
--semihosting ... 454
--silent ... 454
--sockets ... 455
--stlink_reset_strategy ... 455
--suppress_entrypoint_warning ... 456
--timeout ... 456
--xds_board_file ... 456
--xds_reset_strategy ... 457
--xds_rootdir ... 458

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`.

Starting cspybat

1. To start `cspybat` you must first create a batch file. An easy way to do that is to use one of the batch files that C-SPY automatically generates when you start C-SPY in the IDE.

   C-SPY generates a batch file *`projectname`*`.buildconfiguration.cspy.bat` every time C-SPY is initialized. In addition, two more files are generated:

   - *`project.buildconfiguration`*`.general.xcl`, which contains options specific to `cspybat`
   - *`project.buildconfiguration`*`.driver.xcl`, which contains options specific to the C-SPY driver you are using

   You can find the files in the directory `$PROJ_DIR$\settings`. The files contain the same settings as the IDE, and provide hints about additional options that you can use.
2. To start `cspybat`, you can use this command line:

   ```
   project.cspybat.bat [debugfile]
   ```

   Note that *`debugfile`* is optional. You can specify it if you want to use a different debug file than the one that is used in the *`project.buildconfiguration`*`.general.xcl` file.

Output

When you run `cspybat`, these types of output can be produced:

- *Terminal output from cspybat itself*

  All such terminal output is directed to `stderr`. Note that if you run `cspybat` from the command line without any arguments, the `cspybat` version number and all available options including brief descriptions are directed to `stdout` and displayed on your screen.
- *Terminal output from the application you are debugging*

  All such terminal output is directed to `stdout`, provided that you have used the `--plugin` option. See *--plugin, page 450*.
- *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
```



In those cases where a filename is required you are recommended to give a full path to the filename. For the DLL files it is mandatory.

Parameters

The parameters are:

Parameter	Description
processor_DLL	The path to the processor-specific DLL file—available in `arm\bin`.
driver_DLL	The path to the C-SPY driver DLL file—available in `arm\bin`.
debug_file	The object file that you want to debug (filename extension `out`). See also *--debug_file, page 420*.
cspybat_options	The command line options that you want to pass to `cspybat`. Note that these options are optional. For information about each option, see *Reference information on C-SPY command line options, page 416*.
`--backend`	Marks the beginning of the parameters to the C-SPY driver; all options that follow will be sent to the driver. Note that this option is mandatory.
driver_options	The command line options that you want to pass to the C-SPY driver. Note that some of these options are mandatory and some are optional. For information about each option, see *Reference information on C-SPY command line options, page 416*.

Table 52. cspybat parameters

SUMMARY OF C-SPY COMMAND LINE OPTIONS

General cspybat options

--application_args, page 416	Passes command line arguments to the debugged application.
--attach_to_running_target, page 417	Makes the debugger attach to a running application at its current location, without resetting the target system.
--backend, page 417	Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).
--code_coverage_file, page 418	Enables the generation of code coverage information and places it in a specified file.
--cycles, page 418	Specifies the maximum number of cycles to run.
--debug_file, page 420	Specifies an alternative debug file.
--device_macro, page 421	Specifies a C-SPY device macro file.
--download_only, page 422	Downloads a code image without starting a debug session afterwards.
-f, page 432	Extends the command line.
--flash_loader, page 433	Specifies a flash loader specification XML file.
--macro, page 445	Specifies a macro file to be used.
--macro_param, page 446	Assigns a value to a C-SPY macro parameter.
--plugin, page 450	Specifies a plugin file to be used.
--rtc_enable, page 294	Enables C-RUN runtime error checking in `cspybat`.
--rtc_output, page 294	Specifies to `cspybat` a file for the C-RUN message output.
--rtc_raw_to_txt, page 295	Makes `cspybat` act as a runtime checking message filter by reading a file as input.
--rtc_rules, page 295	Specifies a file for the C-RUN rules to `cspybat`.
--silent, page 454	Omits the sign-on message.
--sockets, page 455	Makes the debugger use sockets instead of pipes for its internal communication.

--suppress_entrypoint_warning, page 456	Disables the warning when the ELF entry point is at address `0x0`.
--timeout, page 456	Limits the maximum allowed execution time.

Options available for all C-SPY drivers

`--BE8`	Uses the big-endian format BE8. For reference information, see the *IAR C/C++ Development Guide for Arm*.
`--BE32`	Uses the big-endian format BE32. For reference information, see the *IAR C/C++ Development Guide for Arm*.
`--cpu`	Specifies a processor variant. For reference information, see the *IAR C/C++ Development Guide for Arm*.
--device, page 420	Specifies the name of the device.
--drv_communication, page 423	Specifies the communication link to be used.
--drv_exclude_from_verify, page 427	Excludes memory ranges from being verified.
--drv_reset_to_cpu_start, page 429	Omits setting the `PC` when resetting the application.
--drv_debugger_cache, page 425	Disables memory caching and memory range checking in C-SPY.
--drv_restore_breakpoints, page 430	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 468*, specifically the option **Suppress download**.
--drv_vector_table_base, page 432	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 468*, specifically the option **Verify download**. Available for all hardware drivers.
`--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_target_running, page 445	Starts the execution on the target and then exits but leaves the target running.
-p, page 449	Specifies the device description file to be used.
--proc_stack_stack, page 450	Provides C-SPY with information about reserved stacks.
--semihosting, page 454	Enables semihosted I/O.

Options available for the simulator driver

--disable_interrupts, page 421	Disables the interrupt simulation. (Not available for all cores and devices.)
--function_profiling, page 433	Analyzes your source code to find where the most time is spent during execution. (Not available for all cores and devices.)

--mapu, page 446	Activates memory access checking. (Not available for all cores and devices.)
--multicore_nr_of_cores, page 449	Specifies the number of cores on the device for multicore debugging. (Not available for all cores and devices.)

Options available for the C-SPY GDB Server driver

--drv_default_breakpoint, page 426	Sets the type of breakpoint resource to be used when setting breakpoints.
--drv_enforce_mem_config, page 426	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
--gdbserv_exec_command, page 434	Sends a command string to the GDB Server.

Options available for the C-SPY I-jet driver

--board_file, page 417	Provides C-SPY with information needed for setting flash breakpoints.
--debug_auth_enforce, page 419	Specifies that the authentication procedure should be executed when automatic detection fails.
--debug_auth_settings, page 419	Specifies the XML file that stores parameters required for the authentication.
--debug_auth_type, page 420	Specifies the mechanism used for the debug authentication.
--drv_catch_exceptions, page 422	Makes the application stop for certain exceptions.
--drv_default_breakpoint, page 426	Sets the type of breakpoint resource to be used when setting breakpoints.
--drv_enforce_mem_config, page 426	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
--drv_interface, page 427	Selects the communication interface.
--drv_interface_speed, page 428	Specifies the JTAG and SWD interface speed.
--drv_mem_ap, page 429	Enables Live Watch and Live Memory reads for Cortex-A and Cortex-R devices.
--drv_trace_settings, page 431	Specifies the mechanism and interface for trace data collection.
--jet_board_cfg, page 434	Specifies a probe configuration file.
--jet_board_did, page 434	Selects which CPU to debug on a multicore system.
--jet_cpu_clock, page 435	Specifies the frequency of the internal processor clock.
--jet_disable_pmu, page 436	Disables the PMU event counting.
--jet_disable_pmu_dap, page 436	Disables the PMU DAP channel and uses CP15 instructions for PMU event counting.
--jet_ir_length, page 436	Specifies the number of IR bits preceding the Arm core to connect to.
--jet_power_from_probe, page 437	Specifies the power supply from the I-jet or I-jet Trace probe.

--jet_probe, page 437	Specifies which debug system the C-SPY I-jet driver is an interface to.
--jet_script_file, page 437	Specifies the reset script file.
--jet_standard_reset, page 438	Selects the reset strategy to be used when C-SPY starts.
--jet_startup_connection_timeout, page 439	Prolongs the time that the C-SPY driver tries to connect to the target board.
--jet_swo_on_d0, page 439	Specifies that SWO trace data is output on the trace data pin `D0`.
--jet_swo_prescaler, page 440	Specifies the SWO prescaler for the CPU clock frequency.
--jet_swo_protocol, page 440	Selects the SWO communication protocol.
--jet_tap_position, page 441	Selects a specific device in the JTAG scan chain.
--reset_style, page 451	Specifies the reset strategies that will be available when debugging.
--sdm_debug_architecture, page 452	Specifies the debug architecture used for SDM debug authentication.
--sdm_library, page 452	Specifies the library file used for SDM debug authentication.
--sdm_library_hint, page 453	Specifies an alternate authentication method if the SDM library does not use the standard SDM API.
--sdm_manifest, page 453	Specifies a manifest file that points out the SDM debug authentication library file.

Options available for the C-SPY CMSIS-DAP driver

--debug_auth_enforce, page 419	Specifies that the authentication procedure should be executed when automatic detection fails.
--debug_auth_settings, page 419	Specifies the XML file that stores parameters required for the authentication.
--debug_auth_type, page 420	Specifies the mechanism used for the debug authentication.
--drv_catch_exceptions, page 422	Makes the application stop for certain exceptions.
--drv_default_breakpoint, page 426	Sets the type of breakpoint resource to be used when setting breakpoints.
--drv_enforce_mem_config, page 426	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
--drv_interface, page 427	Selects the communication interface.
--drv_interface_speed, page 428	Specifies the JTAG and SWD interface speed.
--jet_board_cfg, page 434	Specifies a probe configuration file.
--jet_board_did, page 434	Selects which CPU to debug on a multicore system.
--jet_probe, page 437	Specifies which debug system the C-SPY driver is an interface to.
--jet_script_file, page 437	Specifies the reset script file.

--jet_standard_reset, page 438	Selects the reset strategy to be used when C-SPY starts.
--jet_startup_connection_timeout, page 439	Prolongs the time that the C-SPY driver tries to connect to the target board.
--jet_tap_position, page 441	Selects a specific device in the JTAG scan chain.
--reset_style, page 451	Specifies the reset strategies that will be available when debugging.
--sdm_debug_architecture, page 452	Specifies the debug architecture used for SDM debug authentication.
--sdm_library, page 452	Specifies the library file used for SDM debug authentication.
--sdm_library_hint, page 453	Specifies an alternate authentication method if the SDM library does not use the standard SDM API.
--sdm_manifest, page 453	Specifies a manifest file that points out the SDM debug authentication library file.

Options available for the C-SPY J-Link/J-Trace driver

--drv_catch_exceptions, page 422	Makes the application stop for certain exceptions.
--drv_default_breakpoint, page 426	Sets the type of breakpoint resource to be used when setting breakpoints.
--drv_enforce_mem_config, page 426	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
--drv_interface, page 427	Selects the communication interface.
--drv_interface_speed, page 428	Specifies the JTAG and SWD interface speed.
--drv_swo_clock_setup, page 430	Specifies the CPU clock and the wanted SWO speed.
--jlink_dcc_timeout, page 441	Specifies the timeout for a pending request from C-SPY to the DCC agent on target.
--jlink_device_select, page 441	Selects a specific device in the JTAG scan chain.
--jlink_exec_command, page 442	Calls the `__jlinkExecCommand` macro after target connection has been established.
--jlink_initial_speed, page 442	Sets the initial JTAG communication speed in kHz.
--jlink_ir_length, page 443	Sets the number of IR bits preceding the Arm core to connect to.
--jlink_reset_strategy, page 443	Selects the reset strategy to use at debugger startup.
--jlink_script_file, page 444	Specifies the script file for setting up hardware.
--jlink_trace_source, page 444	Selects either ETB or ETM as the trace source.

Options available for the C-SPY TI MSP-FET driver

--drv_enforce_mem_config, page 426	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
--mspfet_erase_flash, page 447	Specifies which flash memory to erase before download.
--mspfet_interface_speed, page 447	Specifies the interface communication speed.

--mspfet_reset_strategy, page 448	Selects the reset strategy to use at debugger startup.
--mspfet_settlingtime, page 448	Specifies a delay that will be used between switching on the target VCC and starting the identification of the Arm device.
--mspfet_vccvoltage, page 448	Specifies the target VCC voltage.

Options available for the C-SPY TI Stellaris driver

--drv_enforce_mem_config, page 426	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
--drv_interface, page 427	Selects the communication interface.
--drv_interface_speed, page 428	Specifies the JTAG and SWD interface speed.
--lmiftdi_reset_strategy, page 445	Selects the reset strategy to use at debugger startup.

Options available for the C-SPY TI XDS driver

--drv_catch_exceptions, page 422	Makes the application stop for certain exceptions.
--drv_default_breakpoint, page 426	Sets the type of breakpoint resource to be used when setting breakpoints.
--drv_enforce_mem_config, page 426	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
--drv_interface, page 427	Selects the communication interface.
--drv_interface_speed, page 428	Specifies the JTAG and SWD interface speed.
--drv_swo_clock_setup, page 430	Specifies the CPU clock and the wanted SWO speed.
--xds_board_file, page 456	Overrides the default board file.
--xds_reset_strategy, page 457	Specifies the reset strategy to use.
--xds_rootdir, page 458	Specifies the installation directory of the TI XDS driver package.

Options available for the C-SPY ST-LINK driver

--debug_auth_enforce, page 419	Specifies that the authentication procedure should be executed when automatic detection fails.
--debug_auth_settings, page 419	Specifies the XML file that stores parameters required for the authentication.
--debug_auth_type, page 420	Specifies the mechanism used for the debug authentication.
--drv_catch_exceptions, page 422	Makes the application stop for certain exceptions.
--drv_enforce_mem_config, page 426	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
--drv_interface, page 427	Selects the communication interface.
--drv_interface_speed, page 428	Specifies the JTAG and SWD interface speed.
--drv_swo_clock_setup, page 430	Specifies the CPU clock and the wanted SWO speed.
--sdm_debug_architecture, page 452	Specifies the debug architecture used for SDM debug authentication.

--sdm_library, page 452	Specifies the library file used for SDM debug authentication.
--sdm_library_hint, page 453	Specifies an alternate authentication method if the SDM library does not use the standard SDM API.
--sdm_manifest, page 453	Specifies a manifest file that points out the SDM debug authentication library file.
--stlink_reset_strategy, page 455	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.

--application_args

Syntax

```
--application_args="arg0 arg1 ..."
```

Parameters

arg	A command line argument.

For use with

`cspybat`

Description

Use this option to pass command line arguments to the debugged application. These variables must be defined in the application:

```
/* __argc, the number of arguments in __argv. */
__no_init __root int __argc;

/* __argv, an array of pointers to the arguments (strings); must be large
   enough to fit the number of arguments.*/
__no_init __root const char * __argv[MAX_ARGS];

/* __argvbuf, a storage area for __argv; must be large enough to hold all
   command line arguments. */
__no_init __root char __argvbuf[MAX_ARG_SIZE];
```

Example

```
--application_args="--logfile log.txt --verbose"
```

This option is not available in the IDE. However, command line arguments can be passed to the debugged application from the **Project>Options>Debugger>Extra Options** page using the `/args` option, see *Extra Options, page 473*.

--attach_to_running_target

Syntax

```
--attach_to_running_target
```

For use with

cspybat



This option might not be supported by the combination of C-SPY driver and device that you are using. If you are using this option with an unsupported combination, C-SPY produces a message.

Description

Use this option to make the debugger attach to a running application at its current location, without resetting the target system.

If you have defined any breakpoints in your project, the C-SPY driver will set them during attachment. If the C-SPY driver cannot set them without stopping the target system, the breakpoints will be disabled. The option also suppresses download and the **Run to** option.



Project>Attach to Running Target

--backend

Syntax

```
--backend {driver options}
```

Parameters

driver options Any option available to the C-SPY driver you are using.

For use with

cspybat (mandatory).

Description

Use this option to send options to the C-SPY driver. All options that follow `--backend` will be passed to the C-SPY driver, and will not be processed by `cspybat` itself.



This option is not available in the IDE.

--board_file

Syntax

```
--board_file filename
```

Parameters

filename The flash loader specification XML file, with the filename extension `board`.

For use with

The C-SPY I-jet driver.

Description

Use this option to provide C-SPY with information needed for setting flash breakpoints. If you do not provide C-SPY with that information, using flash breakpoints will not work.

See also

The *IAR Flash Loader Development Guide*.



To set related options, choose:

Project>Options>Debugger>Download>Use flash loader(s)

--code_coverage_file

Syntax

```
--code_coverage_file=filename
```

Note that this option must be placed before the `--backend` option on the command line.

Parameters

filename	The name of the destination file for the code coverage information.

For use with

`cspybat`

Description

Use this option to enable the generation of a text-based report file for code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file. Because most embedded applications do not terminate, you might have to use this option in combination with `--timeout` or `--cycles`.

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 233, *--cycles, page 418*, *--timeout, page 456*.



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.

--debug_auth_enforce

Syntax

```
--debug_auth_enforce
```

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

The C-SPY ST-LINK driver.

Description

Use this option if, for some reason, the automatic detection mechanism fails and you want to force using the authentication procedure. This could be the case if you are using a device that is not explicitly supported yet by IAR Embedded Workbench.

See also

Debug authentication, page 39.



Project>Options>Debugger>Authentication>Apply authentication if status cannot be determined

--debug_auth_settings

Syntax

```
--debug_auth_settings=path
```

Parameters

path The path to the file where the authentication parameters are stored.

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

The C-SPY ST-LINK driver.

Description

Use this option to specify the XML file that stores parameters required for the authentication. This file is normally the `.dnx` file for the project. In the IDE, this file is selected automatically.

See also

Debug authentication, page 39.



This option is not available in the IDE.

--debug_auth_type

Syntax

```
--debug_auth_type=mechanism
```

Parameters

mechanism	The name of the mechanism used for the debug authentication. Currently, this parameter must be `sdm`.

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver

The C-SPY ST-LINK driver.

Description

Use this option to specify the mechanism used for the debug authentication.

See also

Debug authentication, page 39.



This option is not available in the IDE.

--debug_file

Syntax

```
--debug_file filename
```

Parameters

filename	The name of the debug file to use.

For use with

`cspybat`

Description

Use this option to make `cspybat` use the specified debug file instead of the one used in the generated `cpsybat.bat` file. This option can be placed both before and after the `--backend` option on the command line.



This option is not available in the IDE.

--device

Syntax

```
--device=device_name
```

Parameters

device_name 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

--device_macro

Syntax

```
--device_macro filename
```

Note that this option must be placed before the `--backend` option on the command line.

Parameters

filename The C-SPY device macro file to be used (filename extension `dmac`).

For use with

`cspybat`

Description

Use this option to specify a C-SPY device macro file to be loaded before you execute the target application. A device macro is also loaded when you run a flash loader.

A device macro can include scripted reset styles that can be used by the debugger.

This option can be used more than once on the command line.

See also

Briefly about using C-SPY macros, page 330.

This option is not available in the IDE.

--disable_interrupts

Syntax

```
--disable_interrupts
```

For use with

The C-SPY simulator driver. (Not available for all cores and devices.)

Description

Use this option to disable the interrupt simulation.



To set this option, choose **Simulator>Interrupt Configuration** and deselect the **Enable interrupt simulation** command on the context menu.

--download_only

Syntax

```
--download_only
```

Note that this option must be placed before the `--backend` option on the command line.

For use with

`cspybat`

Description

Use this option to download the code image without starting a debug session afterwards.



Project>Download>Download active application

Alternatively, to set a related option, choose:

Project>Options>Debugger>Setup and deselect **Run to**.

--drv_catch_exceptions

Syntax

```
--drv_catch_exceptions=value
```

Parameters

value	A value in the range of `0-0x1FF`. Each bit specifies which exception to catch:
(for Arm9, Cortex-R4, Arm11, and Cortex-A)	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	A value in the range of `0-0x7FF`. Each bit specifies which exception to catch:
(for Cortex-M)	Bit 0 = CORERESET - 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 I-jet driver.

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

The C-SPY CMSIS-DAP 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

Setting a breakpoint on an exception vector, page 97.



Project>Options>Debugger>*Driver*>Breakpoints>Catch exceptions

--drv_communication

Syntax

```
--drv_communication=connection
```

Parameters

Where `connection` is one of these for the C-SPY GDB Server driver:

Via Ethernet	`TCPIP:ip_address` `TCPIP:ip_address,port` `TCPIP:hostname` `TCPIP:hostname,port` Note that if no port is specified, port `3333` is used by default.

Where `connection` is one of these for the 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 a single probe, and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box. `USB:#select` forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.
Via USB directly to	`USB0-USB3`

the debug probe	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 driver:

Via USB port	`USB:#`*`serial`* where *`serial`* is a string of digits and letters that identifies which probe you want to connect to. The serial number can be found either printed on the probe, or obtained by connecting a single probe, and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box. `USB:#select` forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session.

Where *`connection`* is one of these for the C-SPY ST-LINK driver, the C-SPY TI Stellaris driver, and the C-SPY TI XDS driver:

Via USB port	`USB:#`*`serial`* where *`serial`* is a string of digits and letters that identifies which probe you want to connect to. The serial number can be found either printed on the probe, or obtained by connecting a single probe, and then starting the debug session. The serial number is then displayed in the **Debug Log** window. The serial number is also displayed in the **Debug Probe Selection** dialog box. `USB:#select` forces the **Debug Probe Selection** dialog box to be displayed each time you start a debug session. `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 MSP-FET driver:

Via COM port	`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. If you do not specify the option `--drv_communication`, the debug probe is automatically selected.

For use with

The C-SPY GDB Server driver.

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

The C-SPY ST-LINK driver.

The C-SPY TI MSP-FET driver.

The C-SPY TI Stellaris driver.

The C-SPY TI XDS driver.

Description

Use this option to choose communication link.



Project>Options>Debugger>GDB Server>TCP/IP address or hostname [,port]

Project>Options>Debugger>J-Link/J-Trace>Connection>Communication

Project>Options>Debugger>ST-LINK>Setup

Project>Options>Debugger>TI MSP-FET>Setup

Project>Options>Debugger>TI XDS>Setup

To set this option for the C-SPY TI Stellaris driver, use **Project>Options>Debugger>Extra Options**.

--drv_debugger_cache

Syntax

```
--drv_debugger_cache={on|off}
```

Parameters

`on`	Enables memory caching and memory range checking.
`off`	Disables memory caching and memory range checking.

For use with

All C-SPY drivers.

Description

Use this option to completely disable 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.



***Driver*>Disable Debugger Cache**

--drv_default_breakpoint

Syntax

```
--drv_default_breakpoint={0|1|2}
```

Parameters

`0` Auto (default)

`1` Hardware

`2` Software

For use with

The C-SPY GDB Server driver.

The C-SPY I-jet driver.

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

The C-SPY CMSIS-DAP 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

Breakpoints options, page 112.

Project>Options>Debugger>*Driver*>Breakpoints>Default breakpoint type

--drv_enforce_mem_config

Syntax

```
--drv_enforce_mem_config={on|off}
```

Parameters

`on` Restricts which memory ranges that can be accessed.

`off` Allows C-SPY to access all memory ranges.

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY GDB Server driver.

The C-SPY I-jet driver.

The C-SPY J-Link driver.

The C-SPY ST-Link driver.

The C-SPY TI Stellaris driver.

The C-SPY TI XDS driver.

Description

Use this option to prevent the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.



Driver>Enforce Memory Configuration

--drv_exclude_from_verify

Syntax

```
--drv_exclude_from_verify=startaddr-endaddr
```

Parameters

startaddr	The start of the memory range.
endaddr	The end of the memory range.

For use with

All C-SPY drivers.

Description

Use this option to exclude memory ranges from being verified when the option `--drv_verify_download`, or the option **Verify download** in the **Project>Options** dialog box in the IDE, is used. The option can be specified multiple times to exclude several ranges.



To set this option, use **Project>Options>Debugger>Extra Options**.

--drv_interface

Syntax

```
--drv_interface={SWD|JTAG|cJTAG|auto}
```

Parameters

`SWD`	Specifies the SWD interface.
`JTAG` (default)	Specifies the JTAG interface.
`cJTAG`	Specifies the cJTAG interface. This parameter is only available for the C-SPY I-jet driver, the J-Link/J-Trace driver, and the C-SPY TI XDS driver.
`auto`	Automatically selects the debug interface. This parameter is only available for the C-SPY TI MSP-FET driver.

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

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

The C-SPY ST-LINK driver.

The C-SPY TI MSP-FET driver.

The C-SPY TI Stellaris driver.

The C-SPY TI XDS driver.

Description

Use this option to specify the communication interface between the debug probe and the target system.

The SWD interface uses fewer pins than JTAG. Specify `--drv_interface=SWD` if you want to use the serial-wire output (SWO) communication channel. Alternatively, you can set this option to `JTAG` and also specify the `--jet_swo_on_d0` option. SWO output on Trace_D0 is only supported by the C-SPY I-Jet/I-jet Trace driver.



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 Trace_D0.

See also

- *SWO Trace Window Settings dialog box, page 165*
- *J-Link/J-Trace — Connection, page 494*
- *ST-LINK — Setup, page 497*



Project>Options>Debugger>CMSIS-DAP>Interface/Interface

Project>Options>Debugger>I-jet>Interface>Interface

Project>Options>Debugger>J-Link/J-Trace>Connection>Interface

Project>Options>Debugger>ST-LINK>Setup>Interface

Project>Options>Debugger>TI MSP-FET>Setup>Interface

Project>Options>Debugger>TI Stellaris>Setup>Interface

Project>Options>Debugger>TI XDS>Setup>Interface

--drv_interface_speed

Syntax

```
--drv_interface_speed=kHz
```

Parameters

kHz The frequency in kHz.

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

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

The C-SPY ST-LINK driver.

The C-SPY TI Stellaris driver.

The C-SPY TI XDS driver.

Description

Use this option to set the JTAG and SWD communication speed in kHz.

See also

J-Link/J-Trace — Setup, page 490.



Project>Options>Debugger>CMSIS-DAP>Interface>Interface speed

Project>Options>Debugger>I-jet>Interface>Interface speed

Project>Options>Debugger>J-Link/J-Trace>Setup>JTAG/SWD speed

Project>Options>Debugger>ST-LINK>Setup>Interface>Interface speed

Project>Options>Debugger>TI Stellaris>Setup>JTAG/SWD speed

Project>Options>Debugger>TI XDS>Setup>Interface>Interface speed

--drv_mem_ap

Syntax

```
--drv_mem_ap=n
```

Parameters

n The number of the MEM-AP port that all Live Watch and Live Memory reads will be redirected through.

For use with

The C-SPY I-jet driver.

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

Description

Use this option to enable Live Watch and Live Memory reads for Cortex-A and Cortex-R devices. All Live Watch and Live Memory reads will be redirected through the specified MEM-AP port.



Because the memory is redirected through a MEM-AP port, the memory viewed through this port will not be the same as the memory viewed from the CPU, if the MMU translation is not one to one.



To set this option, use **Project>Options>Debugger>Extra Options**.

--drv_reset_to_cpu_start

Syntax

```
--drv_reset_to_cpu_start
```

For use with

The C-SPY GDB Server driver

The C-SPY I-jet driver

The C-SPY J-Link/J-Trace 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 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
```

Parameters

location An address or function name label.

For use with

The C-SPY GDB Server driver.

The C-SPY I-jet driver.

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

The C-SPY CMSIS-DAP 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 also

Breakpoints options, page 112.



Project>Options>Debugger>*Driver*>Breakpoints>Restore software breakpoints at

--drv_swo_clock_setup

Syntax

```
--drv_swo_clock_setup=frequency,autodetect,wanted
```

Parameters

frequency	The exact clock frequency used by the internal processor clock, HCLK, in Hz. This value is used for configuring the SWO communication speed and for calculating timestamps.
autodetect	0, Manually specified frequency. Use if data packets are lost during transmission. Specify the desired frequency using the parameter *wanted*. 1, Automatic detection. Automatically uses the highest possible frequency that the J-Link debug probe can handle.
wanted	The frequency to be used if *autodetect* is 0, in Hz.

For use with

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

The C-SPY ST-LINK driver.

The C-SPY TI XDS driver.

Description

Use this option to set up the CPU clock. If this option is not used, the CPU clock frequency is by default set to 72 MHz.

Project>Options>Debugger>J-Link/J-Trace>Setup>Clock setup

Project>Options>Debugger>ST-LINK>Communication>Clock setup

Project>Options>Debugger>TI XDS>Communication>Clock setup

--drv_trace_settings

Syntax

```
--drv_trace_settings={Auto|ETB|ETM|SWO|None}
```

Parameters

Auto	Automatically selects the best possible mechanism and interface, depending on probe and board/device capabilities.
ETB	Collects trace data through the on-chip (ETB/MTB) interface.
ETM	Collects trace data through the parallel (ETM) interface.
SWO	Collects trace data through the serial (SWO) interface.
None	Disables trace. In this mode, C-SPY will not access any trace-related on-chip resources.

For use with

The C-SPY I-jet driver.

Description

Use this option to specify the mechanism and interface for trace data collection. For detailed information, see *I-jet — Trace, page 487*.

Project>Options>Debugger>I-jet>Trace>Mode

--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 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 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 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.

The default label for the vector location is:

- `__vector_table` (Cortex-M)
- `__vector` (Cortex-Av7 and Cortex-Rv7)
- `__vector_v8` (Cortex-Av8 and Cortex-Rv8)



Project>Options>Debugger>Extra Options.

-f

Syntax

```
-f filename
```

Parameters

filename	A text file that contains the command line options (default filename extension `xcl`).

For use with

`cspybat`

Description

Use this option to make `cspybat` read command line options from the specified file.

In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character is treated like a space or tab character.

Both C/C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.

This option can be placed either before or after the `--backend` option on the command line.



To set this option, use **Project>Options>Debugger>Extra Options**.

--flash_loader

Syntax

```
--flash_loader filename
```

Note that this option must be placed before the `--backend` option on the command line.

Parameters

filename	The flash loader specification XML file, with the filename extension `board`.

For use with

`cspybat`

Description

Use this option to specify a flash loader specification XML file which contains all relevant information about the flash loading. There can be more than one such argument, in which case each argument will be processed in the specified order, resulting in several flash programming passes.

See also

The *IAR Flash Loader Development Guide*.



To set related options, choose:

Project>Options>Debugger>Download>Use flash loader(s)

--function_profiling

Syntax

```
--function_profiling filename
```

Parameters

filename	The name of the log file where the profiling data is saved.

For use with

The C-SPY simulator driver. (Not available for all cores and devices.)

Description

Use this option to find the functions in your source code where the most time is spent during execution. The profiling information is saved to the specified file. For more information about function profiling, see *Profiling, page 224*.



C-SPY driver>Function Profiling

--gdbserv_exec_command

Syntax

```
--gdbserv_exec_command="string"
```

Parameters

string	String or command sent to the GDB Server. For more information, see the GDB server documentation.

For use with

The C-SPY GDB Server driver.

Description

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



Project>Options>Debugger>Extra Options

--jet_board_cfg

Syntax

```
--jet_board_cfg=probe_configuration_file
```

Parameters

probe_configuration_file	The full path to a probe configuration file.

For use with

The C-SPY I-jet driver.

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>Interface>Probe configuration file

Project>Options>Debugger>I-jet>Interface>Probe configuration file

--jet_board_did

Syntax

```
--jet_board_did={cpu|#cpu_number}
```

Parameters

cpu	If a board configuration file is specified (using `--jet_board_cfg`) and the defined debug system contains more than one CPU, use this parameter to select a CPU. The value

of *cpu* is a text string. The range of valid values are located in the probe configuration file.

`#`*cpunumber* If the debug system is a multicore SWD system, specify the CPU number on the DAP.
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.
Note that `#cpu_number` has no effect if a board configuration file is specified using `--jet_board_cfg`.

For use with

The C-SPY I-jet driver.

The C-SPY CMSIS-DAP driver.

Description

Use this option to specify which CPU to debug on a multicore system.

`--jet_board_did=#cpu_number` is also applicable when `--jet_probe=cmsisdap` is specified.

Example

Selecting the CPU on a multicore device with a probe configuration file:

```
--jet-board-cfg=device.ProbeConfig --jet_board_did=A9_1
```

Selecting the CPU on a multicore device with a JTAG scan chain, where several CPUs are found at the specified TAP position:

```
--jet_tap_position=1 --jet_ir_length=5 --jet_board_did=#2
```

Project>Options>Debugger>CMSIS DAP>Interface>Probe configuration file>CPU

Project>Options>Debugger>CMSIS DAP>Interface>Probe config>Explicit>CPU number on target

Project>Options>Debugger>I-jet>Interface>Probe configuration file>CPU

Project>Options>Debugger>I-jet>Interface>Explicit probe configuration>CPU number on target

--jet_cpu_clock

Syntax

```
--jet_cpu_clock=frequency
```

Parameters

frequency The clock frequency in Hz.

For use with

The C-SPY I-jet 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.

Project>Options>Debugger>I-jet>Trace>CPU clock

--jet_disable_pmu

Syntax

```
--jet_disable_pmu
```

For use with

The C-SPY I-jet driver.

Description

Use this option to disable the PMU event counting completely.

To set this option, use **Project>Options>Debugger>Extra Options**.

--jet_disable_pmu_dap

Syntax

```
--jet_disable_pmu_dap
```

For use with

The C-SPY I-jet driver.

Description

Use this option to disable the PMU DAP channel and revert to using CP15 instructions for PMU event counting. This disables live updating of counter values when the debugger is running.

To set this option, use **Project>Options>Debugger>Extra Options**.

--jet_ir_length

Syntax

```
--jet_ir_length=length
```

Parameters

length	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 driver.

Description

Use this option to set the number of IR bits preceding the Arm core to connect to.

See also

I-jet — Interface, page 485

Project>Options>Debugger>I-jet>Interface>Explicit probe configuration>Preceding bits

--jet_power_from_probe

Syntax

```
--jet_power_from_probe=[leave_on|switch_off]
```

Parameters

`leave_on`	Continues to supply power to the target even after the debug session has been stopped.
`switch_off`	Turns off the power to the target when the debug session stops.

For use with

The C-SPY I-jet driver.

Description

Use this option to specify the status of the probe power supply after debugging.

If this option is not specified, the probe will not supply power to the board.



Project>Options>Debugger>I-jet>Setup>Target power

--jet_probe

Syntax

```
--jet_probe=[ijet|cmsisdap]
```

Parameters

`ijet`	Specifies the C-SPY I-jet driver as the interface to an I-jet or I-jet Trace probe.
`cmsisdap`	Specifies the C-SPY I-jet driver as the interface to a CMSIS-DAP system.

For use with

The C-SPY I-jet driver.

The C-SPY CMSIS-DAP driver.

Description

Use this option to specify the C-SPY I-jet driver as the interface to a debug system.



Project>Options>Debugger>Driver

--jet_script_file

Syntax

```
--jet_script_file=path
```

Parameters

path	The path to the file where the scripted reset strategies are described.

For use with

The C-SPY I-jet 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 451 and *--jet_standard_reset, page 438*.



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.

delay The delay time, in milliseconds, after the reset signal has been de-asserted, before the debugger attempts to control the processor.

The processor might be kept internally in reset for some time after the external reset signal has been de-asserted, and thus inaccessible for the debugger.

This parameter applies to the hardware reset, and to those custom reset strategies that use the Hardware reset.

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

Description

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.

See also

--reset_style, page 451 and *--jet_script_file, page 437*.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.



Project>Options>Debugger>CMSIS DAP>Setup>Reset

Project>Options>Debugger>I-jet>Setup>Reset

--jet_startup_connection_timeout

Syntax

```
--jet_startup_connection_timeout=milliseconds
```

Parameters

milliseconds	The time in milliseconds.

For use with

The C-SPY I-jet driver.

The C-SPY CMSIS-DAP driver.

Description

Use this option to prolong the time that the C-SPY driver tries to connect to the target board.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.



To set this option, use **Project>Options>Debugger>Extra Options**.

--jet_swo_on_d0

Syntax

```
--jet_swo_on_d0
```

For use with

The C-SPY I-jet 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>Trace>SWO on the TraceD0 pin

--jet_swo_prescaler

Syntax

```
--jet_swo_prescaler=number
```

Parameters

number	The prescaler value, 1–100, which in turn determines the CPU clock frequency.

For use with

The C-SPY I-jet 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.



Project>Options>Debugger>I-jet>Trace>SWO prescaler

--jet_swo_protocol

Syntax

```
--jet_swo_protocol={auto|Manchester|UART}
```

Parameters

`auto`	Selects the communication protocol automatically.
`Manchester`	Specifies the Manchester protocol.
`UART`	Specifies the UART protocol.

For use with

The C-SPY I-jet driver.

Description

Use this option to specify the communication protocol for the SWO channel. If this option is not specified, `auto` is automatically used.



Project>Options>Debugger>I-jet>Trace>SWO protocol

--jet_tap_position

Syntax

```
--jet_tap_position=tap_number|multidrop_id
```

Parameters

tap_number	The TAP position of the device you want to connect to.
multidrop_id	The target ID in a multi-drop system.

For use with

The C-SPY I-jet 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-jet — Interface, page 485.

Project>Options>Debugger>I-jet>Interface>Explicit probe configuration>Target number (TAP or Multidrop ID)

--jlink_dcc_timeout

Syntax

```
--jlink_dcc_timeout=milliseconds
```

Parameters

milliseconds	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.

User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

To set this option, use **Project>Options>Debugger>Extra Options**.

--jlink_device_select

Syntax

```
--jlink_device_select=tap_number
```

Parameters

tap_number 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-jet — Interface, page 485.



Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>TAP number

--jlink_exec_command

Syntax

```
--jlink_exec_commmand=cmdstr1; cmdstr2; cmdstr3 ...
```

Parameters

cmdstrn 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 `__jlinkExecCommand` macro with one or several command strings, after target connection has been established.

See also

__jlinkExecCommand, page 363.



To set this option, use **Project>Options>Debugger>Extra Options**.

--jlink_initial_speed

Syntax

```
--jlink_initial_speed=speed
```

Parameters

speed 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

J-Link/J-Trace — Setup, page 490.



Project>Options>Debugger>J-Link/J-Trace>Setup>JTAG speed>Fixed

--jlink_ir_length

Syntax

```
--jlink_ir_length=length
```

Parameters

length	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

J-Link/J-Trace — Connection, page 494.



Project>Options>Debugger>J-Link/J-Trace>Connection>JTAG scan chain>Preceding bits

--jlink_reset_strategy

Syntax

```
--jlink_reset_strategy=delay,strategy
```

Parameters

delay	For Cortex-M and Arm 7/9/11 with strategies `1-9`, *delay* should be `0` (ignored). For Arm 7/9/11 with strategy `0`, the delay should be one of `0-10000`.
strategy	For information about supported reset strategies, see the J-Link/J-Trace documentation available at *www.segger.com*.

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

J-Link/J-Trace — Setup, page 490.



Project>Options>Debugger>J-Link/J-Trace>Setup>Reset

--jlink_script_file

Syntax

```
--jlink_script_file=filename
```

Parameters

filename	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

For a detailed description of the script language. see the J-Link/J-Trace documentation available at *www.segger.com* .



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.



This option only applies to J-Trace.

See also

J-Link/J-Trace — Setup, page 490.



Project>Options>Debugger>J-Link/J-Trace>Setup>ETM/ETB

--leave_target_running

Syntax

```
--leave_target_running
```

For use with

`cspybat`.

Any C-SPY hardware debugger driver.



Even if this option is supported by the C-SPY driver you are using, there might be device-specific limitations.

Description

Use this option to make the debugger leave the application running on the target hardware after the debug session is closed.



Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this option.



C-SPY driver>Leave Target Running

--lmiftdi_reset_strategy

Syntax

```
--lmiftdi_reset_strategy=delay,strategy
```

Parameters

delay	The delay time measured in milliseconds. *delay* is ignored and should be `0`.
strategy	The reset strategy: `0`, (Software) Resets the core. This sets `PC` to the program entry address and `SP` to the initial stack pointer value. `1`, (System) Resets the core and peripheral units. This sets `PC` to the program entry address and `SP` to the initial stack pointer value.

For use with

The C-SPY TI-Stellaris driver.

Description

Use this option to select the reset strategy to be used at debugger startup.

See also

TI Stellaris — Setup, page 502



Project>Options>Debugger>TI Stellaris>Setup>Reset

--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 330.



Project>Options>Debugger>Setup>Setup macros>Use macro file

--macro_param

Syntax

```
--macro_param [param=value]
```

Note that this option must be placed before the `--backend` option on the command line.

Parameters

`param=value` *`param`* is a parameter defined using the `__param` C-SPY macro construction. `value` is a value.

For use with

`cspybat`

Description

Use this option to assign a value to a C-SPY macro parameter. This option can be used more than once on the command line.

See also

Macro parameters, page 335.



To set this option, use **Project>Options>Debugger>Extra Options**

--mapu

Syntax

```
--mapu
```

For use with

The C-SPY simulator driver. (Not available for all cores and devices.)

Description

Specify this option to use the section information in the debug file for memory access checking. During the execution, the simulator will then check for accesses to unspecified memory ranges. If any such access is found, the C function call stack and a message will be printed on `stderr` and the execution will stop.

See also

Monitoring memory and registers, page 123.



To set related options, choose:

Simulator>Memory Access Setup

--mspfet_erase_flash

Syntax

```
--mspfet_erase_flash=main|main_info|main_info_ip
```

Parameters

`main`	Erases the main flash memory.
`main_info`	Erases both flash memories—main and Information memory.
`main_info_ip`	Erases the main and Information flash memories, including the IP protected area.

For use with

The C-SPY TI MSP-FET driver.

Description

Use this option to specify which flash memories to erase before download.



Project>Options>Debugger>TI MSP-FET>Download>Flash erase

--mspfet_interface_speed

Syntax

```
--mspfet_interface_speed=fast|medium|slow
```

Parameters

`fast`	The fast interface speed.
`medium`	The medium interface speed.
`slow`	The slow interface speed.

For use with

The C-SPY TI MSP-FET driver.

Description

Use this option to set the interface communication speed.



Project>Options>Debugger>TI MSP-FET>Setup>Interface speed

--mspfet_reset_strategy

Syntax

```
--mspfet_reset_strategy=delay,strategy
```

Parameters

delay	The delay time in milliseconds.
strategy	The reset strategy, where `0` is Normal.

For use with

The C-SPY TI MSP-FET driver.

Description

Use this option to select the reset strategy to use at debugger startup.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.



Project>Options>Debugger>TI MSP-FET>Setup>Reset

--mspfet_settlingtime

Syntax

```
--mspfet_settlingtime=delay
```

Parameters

delay	The delay in milliseconds.

For use with

The C-SPY TI MSP-FET driver.

Description

Use this option to specify a delay that will be used between switching on the target VCC and starting the identification of the Arm device.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.



Project>Options>Debugger>TI MSP-FET>Setup>Target VCC>Settling time

--mspfet_vccvoltage

Syntax

```
-mspfet_vccvoltage=voltage
```

Parameters

voltage	The target VCC voltage in mV.

For use with

The C-SPY TI MSP-FET driver.

Description

Use this option to specify the target VCC voltage.



Project>Options>Debugger>TI MSP-FET>Setup>Target VCC>Target VCC

--multicore_nr_of_cores

Syntax

```
--multicore_nr_of_cores=cores
```

Parameters

cores	The number of cores on your device. This must be an integer from 2–8.

For use with

The C-SPY simulator driver. (Not available for all cores and devices.)

Description

For symmetric multicore debugging, specify the number of cores on your device. This option is not needed for debugging a single-core system, or for asymmetric multicore debugging.

See also

Multicore debugging, page 298.



Project>Options>Debugger>Multicore>Number of cores

-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 35.



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 `armLibsupportUniversal.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.



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



Project>Options>Debugger>Plugins

--proc_stack_*stack*

Syntax

```
--proc_stack_stack=startaddress,endaddress
```

where *stack* is:

- one of `main` or `proc` for Cortex-M, *and*
- one of `usr`, `svc`, `irq`, `fiq`, `und`, or `abt` for other Arm cores

Parameters

startaddress	The start address of the stack, specified either as a value or as an expression.
endaddress	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 `cspybat`.

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

```
--proc_stack_irq=0x8000,0x80FF
```



To set this option, use **Project>Options>Debugger>Extra Options**.

--reset_style

Syntax

```
--reset_style="reset_id,reset_name,selected,menu_command"
```

Parameters

reset_id	The number of the reset strategy, `0-9`, as described for `--jet_standard_reset`
reset_name	The name of the reset strategy, according to the file specified by `--jet_script_file`. For the built-in reset strategies, this parameter is `-`. To override a built-in reset strategy, enter the label or function name in your reset script file.
selected	`0` or `1`, where `1` sets the default reset strategy for the **Reset** drop-down menu
menu_command	The name of the reset strategy as it will be displayed on the **Reset** drop-down menu.

For use with

The C-SPY I-jet driver.

The C-SPY CMSIS-DAP driver.

Description

Use this option to specify the reset strategies that will be available when debugging, once for each reset strategy.

Example

This example specifies a script file, sets the standard reset strategy, and specifies the reset strategies that will be available when debugging:

```
--jet_script_file=myDir\myProbeScriptFile
--jet_standard_reset=9,0,0
--reset_style="0,-,0,Disabled (no reset)"
--reset_style="1,-,0,Software"
--reset_style="2,-,0,Hardware"
--reset_style="3,-,0,Core"
--reset_style="4,-,0,System"
--reset_style="5,Custom,0,Custom reset"
--reset_style="9,ConnectUnderReset,1,Connect during reset"
```

See also

--jet_script_file, page 437 and *--jet_standard_reset, page 438*



To set this option, use **Project>Options>Debugger>Extra Options**.

--sdm_debug_architecture

Syntax

```
--sdm_debug_architecture=interface
```

Parameters

interface	The SDM debug architecture used for SDM debug authentication. Choose between: `adi_v5` — Arm Debug Interface V5 (default) `adi_v6` — Arm Debug Interface V6 `nexus` — the Nexus (IEEE-ISTO 5001-2003) standard debug interface

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

The C-SPY ST-LINK driver.

Description

Use this option to specify the debug architecture used for SDM debug authentication.

See also

Debug authentication, page 39.



To set this option, use **Project>Options>Debugger>Extra Options**.

--sdm_library

Syntax

```
--sdm_library=path
```

Parameters

path	The path to the SDM library file (`.dll`).

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

The C-SPY ST-LINK driver.

Description

Use this option to specify the path to the library file (`.dll`) used for SDM debug authentication.



This option cannot be used together with the option `--sdm_manifest`.

See also

Debug authentication, page 39 and *--sdm_manifest, page 453*.



Project>Options>Debugger>Authentication>Library file.

--sdm_library_hint

Syntax

```
--sdm_library_hint=ID
```

Parameters

ID An implementation-specific identifier.

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

The C-SPY ST-LINK driver.

Description

Use this option if the SDM library does not use the standard SDM API. In that case, IAR Embedded Workbench must handle the authentication in a special way. For the STM32H5 device family and the corresponding library, the identifier is `st_sdm`.

In the IDE, the implementation-specific identifier is fetched from the device selection file (`.i79`).

See also

Debug authentication, page 39.



This option is not available in the IDE.

--sdm_manifest

Syntax

```
--sdm_manifest=path
```

Parameters

path The path to the SDM manifest file (`.xml`).

For use with

The C-SPY CMSIS-DAP driver.

The C-SPY I-jet driver.

The C-SPY ST-LINK driver.

Description

Use this option to specify the path to the manifest file (`.xml`) that points out the library file used for SDM debug authentication.



This option cannot be used together with the option `--sdm_library`.

See also

Debug authentication, page 39 and *--sdm_library, page 452*.



Project>Options>Debugger>Authentication>Manifest file.

--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

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.

To make semihosting work, your application must be linked with a semihosting library.

* If no parameter is specified, the behavior is as described in .the chapter *The DLIB runtime environment* in the *IAR C/C++ Development Guide for Arm*.

See also

For information about linking with semihosting, see .the chapter *The DLIB runtime environment* in the *IAR C/C++ Development Guide for Arm*.



Project>Options>General Options>Library Configuration

--silent

Syntax

```
--silent
```

Note that this option must be placed before the `--backend` option on the command line.

For use with

`cspybat`

Description

Use this option to omit the sign-on message.



This option is not available in the IDE.

--sockets

Syntax

```
--sockets
```

Note that this option must be placed before the `--backend` option on the command line.

For use with

`cspybat`

Description

Use this option to make the debugger use sockets instead of pipes for its internal communication. This can make the debugger work better when you run several cspybat instances in parallel on the same host computer.



This option is not available in the IDE.

--stlink_reset_strategy

Syntax

```
--stlink_reset_strategy=delay,strategy
```

Parameters

delay	The delay time measured in milliseconds. *delay* is ignored and should be `0`.
strategy	The reset strategy: `0`, (System) performs the standard reset procedure. Only available for debugging Cortex-M devices. `1`, (Hardware) 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 and when debugging Cortex-M devices. `3`, (Core) resets the core via the `VECTRESET` (Cortex-M) or `CWRR` (Cortex-A) bit—the peripheral units are not affected. `4`, (Software) Sets the `PC` to program entry address and `SP` to the initial stack pointer value.

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

ST-LINK — Setup, page 497



Project>Options>Debugger>ST-LINK>Setup>Reset

--suppress_entrypoint_warning

Syntax

```
--suppress_entrypoint_warning
```

For use with

`cspybat`

Description

Use this option to disable the warning in the debug log when a debug session starts with the ELF entry point at address `0x0` (which according to the ELF standard means that there is no entry point). In practice, there is no problem with having the entry point at address `0x0`.



To set this option, use **Project>Options>Debugger>Extra Options**

--timeout

Syntax

```
--timeout milliseconds
```

Note that this option must be placed before the `--backend` option on the command line.

Parameters

milliseconds	The number of milliseconds before the execution stops.

For use with

`cspybat`

Description

Use this option to limit the maximum allowed execution time.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.



This option is not available in the IDE.

--xds_board_file

Syntax

```
--xds_board_file=dat_file
```

Parameters

dat_file 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

```
--xds_reset_strategy=delay,strategy
```

Parameters

delay The delay time measured in milliseconds.

strategy 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

TI XDS — Setup, page 503



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.



Project>Options>Debugger>TI XDS>Setup>Reset

--xds_rootdir

Syntax

```
--xds_rootdir=path
```

For use with

The C-SPY TI XDS driver.

Description

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 `XDS_EMUPACK_DIR` to set a new default value.



To set this option, use **Project>Options>Debugger>TI XDS>Setup>TI emulation package installation path**.

Flash loaders

Contents

Introduction to the flash loader 459
Using flash loaders 459
Setting up the flash loader(s) 459
The flash loading mechanism 459
Aborting a flash loader 460
Reference information on the flash loader 460
Flash Loader Overview dialog box 460
Flash Loader Configuration dialog box 462

INTRODUCTION TO THE FLASH LOADER

A flash loader is an agent that is downloaded to the target. It fetches your application from the debugger and programs it into flash memory. The flash loader uses the file I/O mechanism to read the application program from the host. You can select one or several flash loaders, where each flash loader loads a selected part of your application. This means that you can use different flash loaders for loading different parts of your application.

Flash loaders for various microcontrollers are provided with IAR Embedded Workbench for 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

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 460*. 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 information about the various flash loader options, see *Flash Loader Configuration dialog box, page 462*.

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

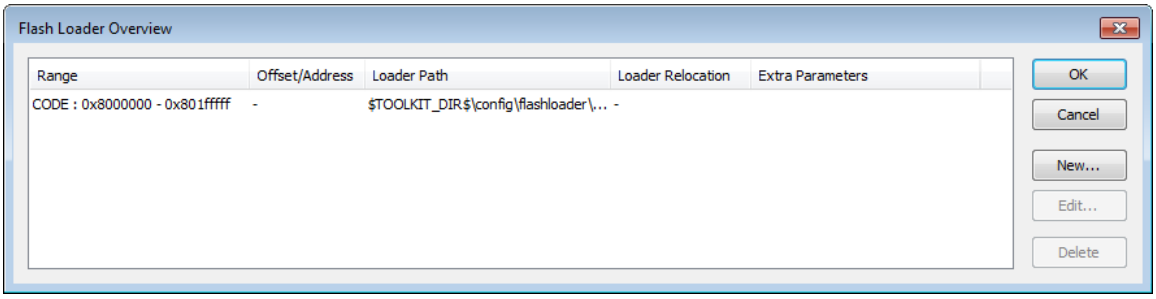
To abort a flash loader:

1. Press Ctrl+Shift+. (period) for a short while.
2. A message that says that the flash loader has aborted is displayed in the **Debug Log** window.
   This method can be used if you suspect that something is wrong with the execution, for example because it seems not to terminate in a reasonable time.

REFERENCE INFORMATION ON THE FLASH LOADER

Flash Loader Overview dialog box

The **Flash Loader Overview** dialog box is available from the **Project>Options>Debugger>Download** page.



This dialog box lists all defined flash loaders. If you have selected a device on the **Project>Options>General Options>Target** page for which there is a flash loader, this flash loader is by default listed in the **Flash Loader Overview** dialog box.

Requirements

Available for supported hardware debugger systems.

Display area

Each row in the display area shows how you have set up one flash loader for flashing a specific part of memory:

Range The part of your application to be programmed by the selected flash loader.

Offset/Address The start of the memory where your application will be flashed. If the address is preceded by an `A`, the address is absolute. Otherwise, it is a relative offset to the start of the memory.

Loader Path The path to the flash loader `*.flash` file to be used (`*.out` for old-style flash loaders).

Loader Relocation For relocatable flash loaders, this is the start of the target RAM memory where the flash loader will be downloaded.

Extra Parameters List of extra parameters that will be passed to the flash loader.

Click on the column headers to sort the list by range, offset/address, etc.

Function buttons

These function buttons are available:

OK The selected flash loader(s) will be used for downloading your application to memory.

Cancel Standard cancel.

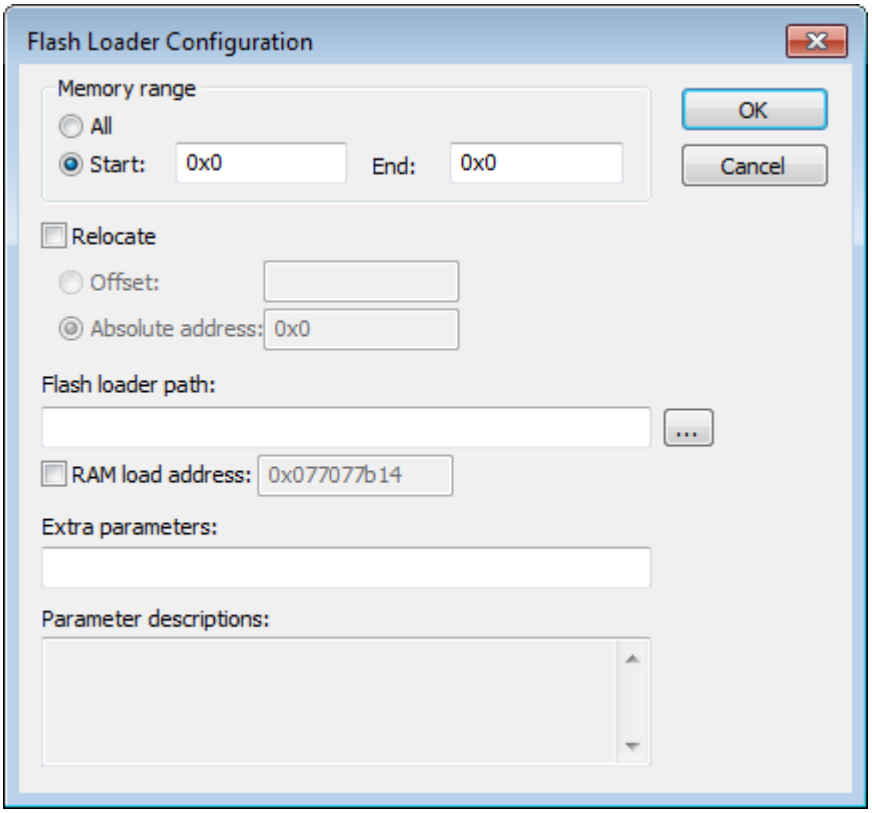
New Displays a dialog box where you can specify what flash loader to use, see *Flash Loader Configuration dialog box, page 462*.

Edit Displays a dialog box where you can modify the settings for the selected flash loader, see *Flash Loader Configuration dialog box, page 462*.

Delete Deletes the selected flash loader configuration.

Flash Loader Configuration dialog box

The **Flash Loader Configuration** dialog box is available from the **Flash Loader Overview** dialog box.



Use the **Flash Loader Configuration** dialog box to configure the download to suit your board. A copy of the default `board` file will be created in your project directory.

Requirements

Available for supported hardware debugger systems.

Memory range

Specify the part of your application to be downloaded to flash memory. Choose between:

All The whole application is downloaded using this flash loader.

Start/End Specify the start and the end of the memory area for which part of the application will be downloaded.

Relocate

Overrides the default flash base address, in other words, relocates the location of the application in memory. This means that you can flash your application to a different location from where it was linked. Choose between:

Offset	A numeric value for a relative offset. This offset will be added to the addresses in the application file.
Absolute address	A numeric value for an absolute base address where the application will be flashed. The lowest address in the application will be placed on this address. Note that you can only use one flash loader for your application when you specify an absolute address.

You can use these numeric formats:

- `123456`, decimal numbers
- `0x123456`, hexadecimal numbers
- `0123456`, octal numbers

The default base address used for writing the first byte—the lowest address—to flash is specified in the linker configuration file used for your application. However, it can sometimes be necessary to override the flash base address and start at a different location in the address space. This can, for example, be necessary for devices that remap the location of the flash memory.

Flash loader path

Use the text box to specify the path to the flash loader file (`*.flash`) to be used by your board configuration.

RAM load address

If the flash loader is relocatable, this option overrides the default address in the target RAM memory that flash loader is downloaded to, in other words, relocates the flash loader. Use the text box to specify the address.

Extra parameters

Some flash loaders define their own set of specific options. Use this text box to specify options to control the flash loader. For information about available flash loader options, see the **Parameter descriptions** field.

Parameter descriptions

Displays a description of the extra parameters specified in the **Extra parameters** text box.

Part 4. Additional reference information

Part 4. Additional reference information

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

- *Debugger options, page 465*
- *Additional information on C-SPY drivers, page 509*

Debugger options

Contents

Setting debugger options 465
Reference information on general debugger options 467
Setup 467
Download 468
Images 469
Multicore 470
Authentication 471
Extra Options 473
Plugins 474
Reference information on C-SPY hardware debugger driver options 475
CADI — Setup 475
CMSIS-DAP — Setup 476
CMSIS-DAP — Interface 479
E2 — Setup 480
GDB Server 481
G+LINK — Setup 482
I-jet — Setup 482
I-jet — Interface 485
I-jet — Trace 487
J-Link/J-Trace — Setup 490
J-Link/J-Trace — Connection 494
Nu-Link — Setup 495
PE micro — Setup 496
ST-LINK — Setup 497
ST-LINK — Communication 499
ST-LINK — Multicore 500
TI MSP-FET — Setup 500
TI MSP-FET — Download 502
TI Stellaris — Setup 502
TI XDS — Setup 503
TI XDS — Communication 505
Third-Party Driver options 507

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 467*.
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 CADI driver	*CADI — Setup, page 475*
C-SPY CMSIS-DAP driver	*CMSIS-DAP — Setup, page 476* *CMSIS-DAP — Interface, page 479* *Breakpoints options, page 112*
C-SPY E2 driver	*E2 — Setup, page 480*
C-SPY GDB Server driver	*GDB Server, page 481* *Breakpoints options, page 112*
C-SPY G+LINK driver	*G+LINK — Setup, page 482*
C-SPY I-jet driver	*I-jet — Setup, page 482* *I-jet — Interface, page 485* *I-jet — Trace, page 487* *Breakpoints options, page 112*
C-SPY J-Link/J-Trace driver	*J-Link/J-Trace — Setup, page 490* *J-Link/J-Trace — Connection, page 494* *Breakpoints options, page 112*
C-SPY Nu-Link driver	*Nu-Link — Setup, page 495*
C-SPY PE micro driver	*PE micro — Setup, page 496*
C-SPY ST-LINK driver	*ST-LINK — Setup, page 497* *ST-LINK — Communication, page 499* *ST-LINK — Multicore, page 500* *Breakpoints options, page 112*
C-SPY TI MSP-FET driver	*TI MSP-FET — Setup, page 500* *TI MSP-FET — Download, page 502*
C-SPY TI Stellaris driver	*TI Stellaris — Setup, page 502*
C-SPY TI XDS driver	*TI XDS — Setup, page 503* *TI XDS — Communication, page 505*
Third-party driver	*Third-Party Driver options, page 507*.

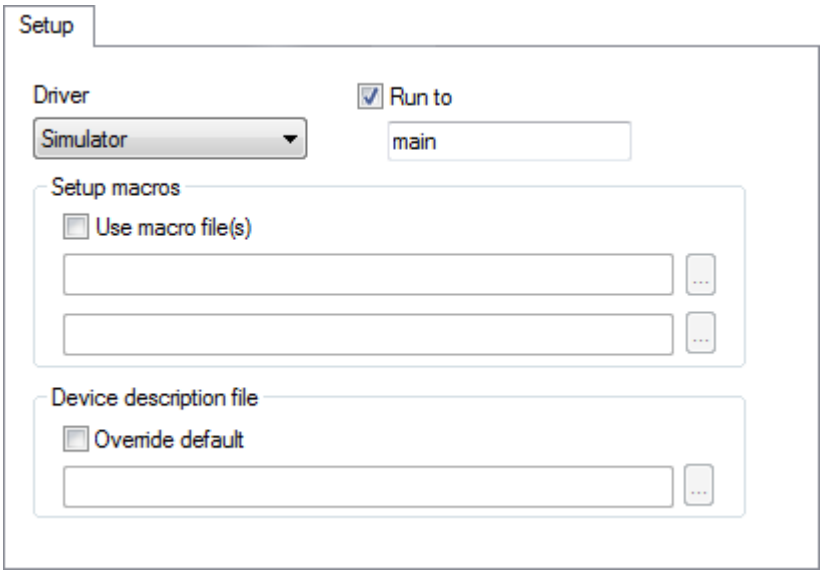
Table 53. Options specific to the C-SPY drivers you are using

5. To restore all settings to the default factory settings, click the **Factory Settings** button.
6. When you have set all the required options, click **OK** in the **Options** dialog box.

REFERENCE INFORMATION ON GENERAL DEBUGGER OPTIONS

Setup

The general **Setup** options select the C-SPY driver, the setup macro file, and device description file to use, and specify which default source code location to run to.



Driver

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

Run to

Specifies the location C-SPY runs to when the debugger starts after a reset. By default, C-SPY runs to the `main` function.

To override the default location, specify the name of a different location you want C-SPY to run to. You can specify assembler labels or whatever can be evaluated as such, for example function names.

If the option is deselected, the program counter will contain the regular hardware reset address at each reset.

See also *Executing from reset, page 34*.

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.

It is possible to specify up to two different macro files.

Device description file

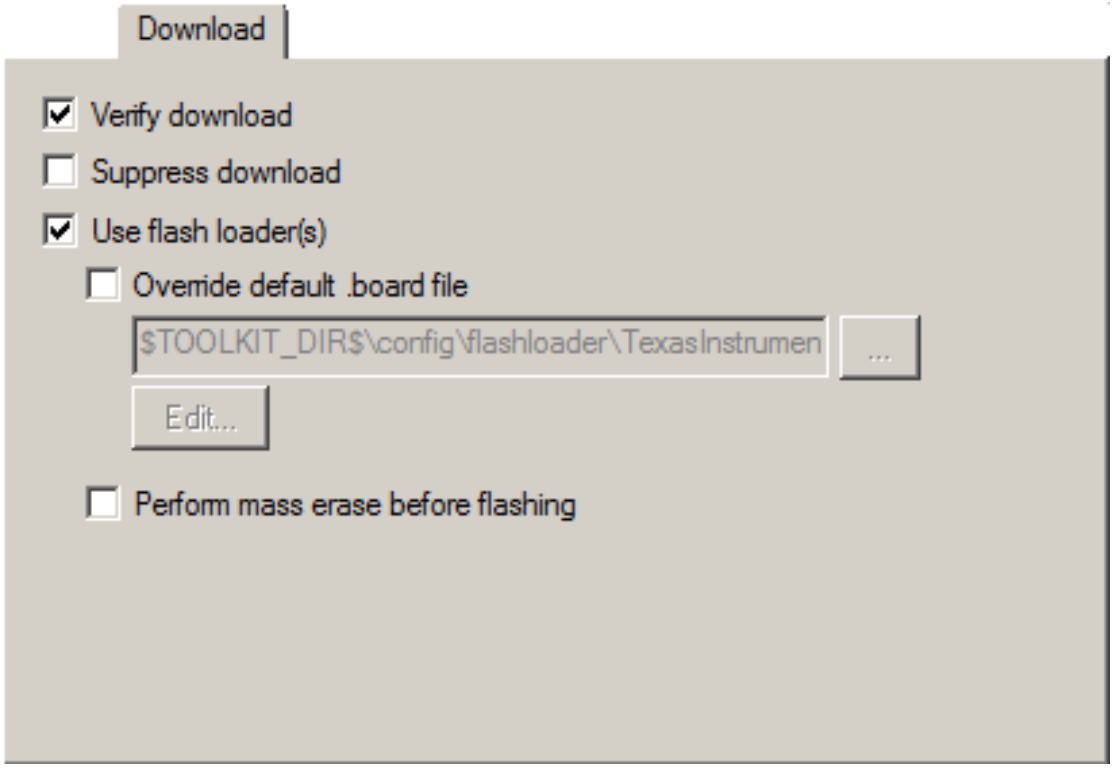
A default device description file—either an IAR-specific `ddf` file or a CMSIS System View Description file—is selected automatically based on your project settings. To override the default file, select **Override default** and specify an alternative file. A browse button is available.

For information about the device description file, see *Modifying a device description file, page 38*.

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.



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.

See *Flash loaders, page 459*.

Override default .board file

A default flash loader is selected based on your choice of device on the **General Options>Target** page. To override the default flash loader, select **Override default .board file** and specify the path to the flash loader you want to use. A browse button is available. Click **Edit** to display the **Flash Loader Overview** dialog box, see *Flash Loader Overview dialog box, page 460*.

Perform mass erase before flashing

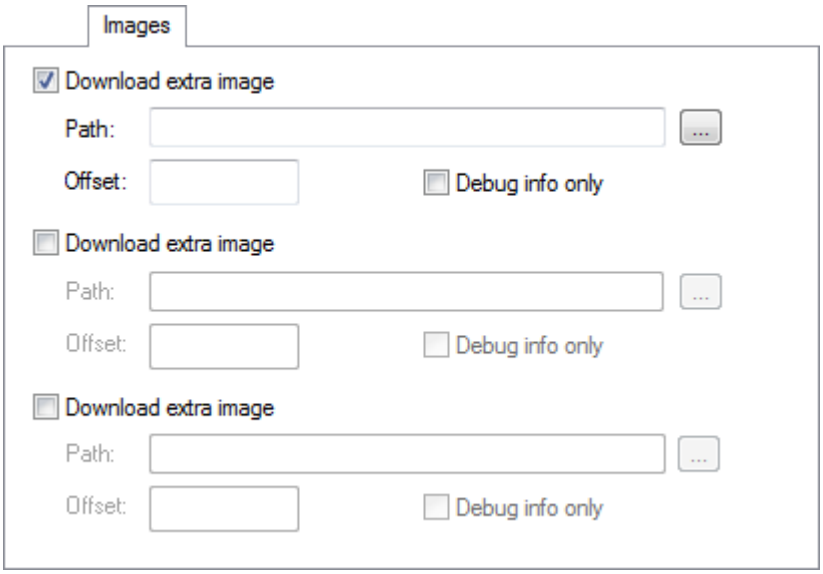
Use this option to perform a mass erase on your device. Mass erase uses an on-chip algorithm that is more efficient than erasing using the flash loader.



This option is only available if your flash loader supports mass erase.

Images

The **Images** options control the use of additional debug files to be downloaded.



Download extra Images

Controls the use of additional debug files to be downloaded:

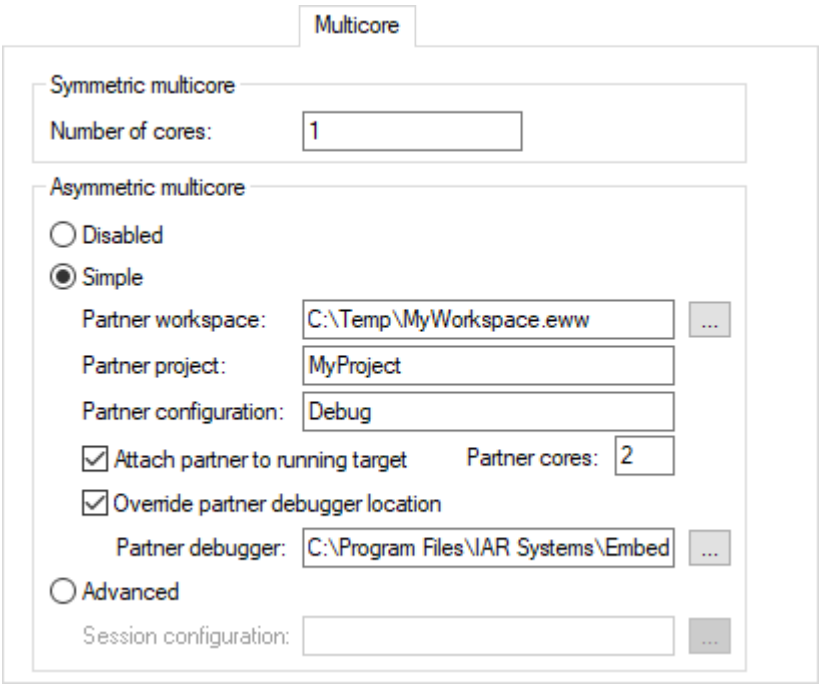
Path	Specify the debug file to be downloaded. A browse button is available.
Offset	Specify an integer that determines the destination address for the downloaded debug file.
Debug info only	Makes the debugger download only debug information, and not the complete debug file.

If you want to download more than three debug images, use the related C-SPY macro, see *__loadImage, page 368*.

For more information, see *Loading multiple debug images, page 36*.

Multicore

The **Multicore** options configure multicore debugging.



Number of cores

For symmetric multicore debugging, specify the number of cores on your device. For asymmetric multicore debugging, specify the number of cores in the master project. (Normally, this will be `1`.)

Disabled

Selecting this option makes the debug session symmetric multicore.

Simple

Selecting this option makes the debug session an asymmetric multicore debugger master. When you start a debug session, a new instance of the IAR Embedded Workbench IDE will be started, using the following options:

Partner workspace Specify the workspace to be opened in the partner (slave) instance.

Partner project Specify the name of the project in the workspace to be opened in the partner instance. For example, if the project filename is `MyPartnerProj.ewp`, specify `MyPartnerProj`.

Partner configuration Specify the build configuration to be used when debugging the partner. For example, `Debug` or `Release`.

Attach partner to running target If you have selected the command **Attach to Running Target** from the **Project** menu, which affects the master. You can also select **Attach partner to running target** to also make the debugger attach the partner to the running application at its current location, without resetting the target system.

For information about **Attach to Running Target**, see the *IDE Project Management and Building Guide for Arm*.

Partner cores Specify the number of cores in the partner project.

Override partner debugger location If the Embedded Workbench instance associated with the *partner* project is not installed in the same location as the Embedded Workbench instance associated with the *master* project, for example in `c:\iar\ewarm-x.x`, you must specify the installation directory of the Embedded Workbench for the partner project. Note that the Embedded Workbench must be based on version 9.1.7 or later of the shared components—to check this, choose **Help>About>Product Info**.

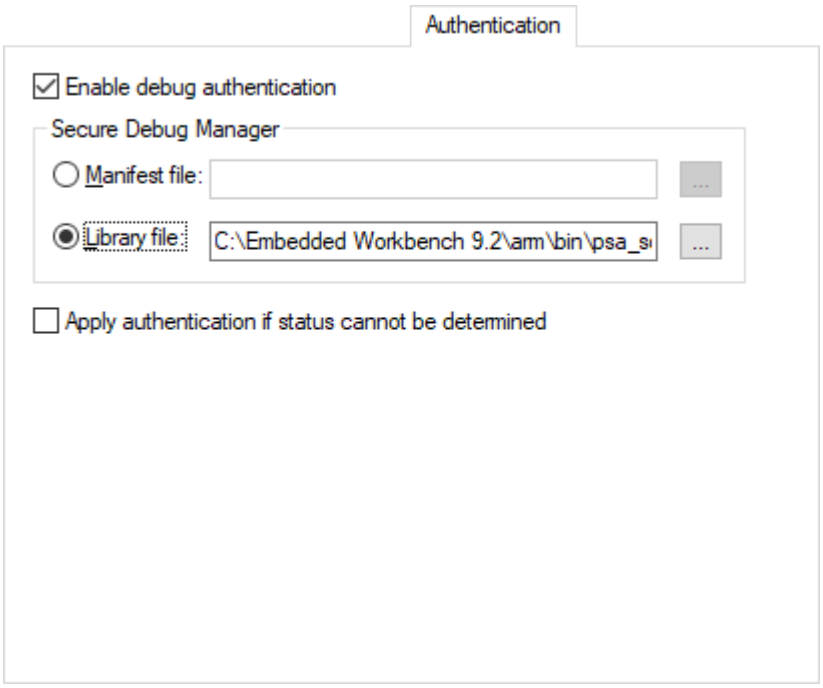
Advanced

Selecting this option makes the debug session asymmetric multicore with one or more new instances of the IAR Embedded Workbench IDE.

Session configuration Use the browse button to specify the XML multicore session file that contains the settings for the debug session. For more information about this file, see *The multicore session file, page 304*.

Authentication

The **Authentication** options configure debug authentication.



Debug authentication reduces the risk that an attacker uses debug capabilities to compromise the target system. IAR Embedded Workbench supports debug authentication using the open-source Secure Debug Manager (SDM) mechanism.

For more information, see *Debug authentication, page 39*.

Enable debug authentication

Enables/disables the debug authentication mechanism. If authentication is enabled, it is performed when the debug session starts. The authentication is revoked by a target reset. The type of reset that revokes the authentication is hardware-specific.

Secure Debug Manager

Use one of these options to specify the authentication library file that corresponds to your target device. This library is needed for the SDM debug authentication. Choose between:

Manifest file Specify the location of a manifest file (`.xml`) that points out the actual library file.

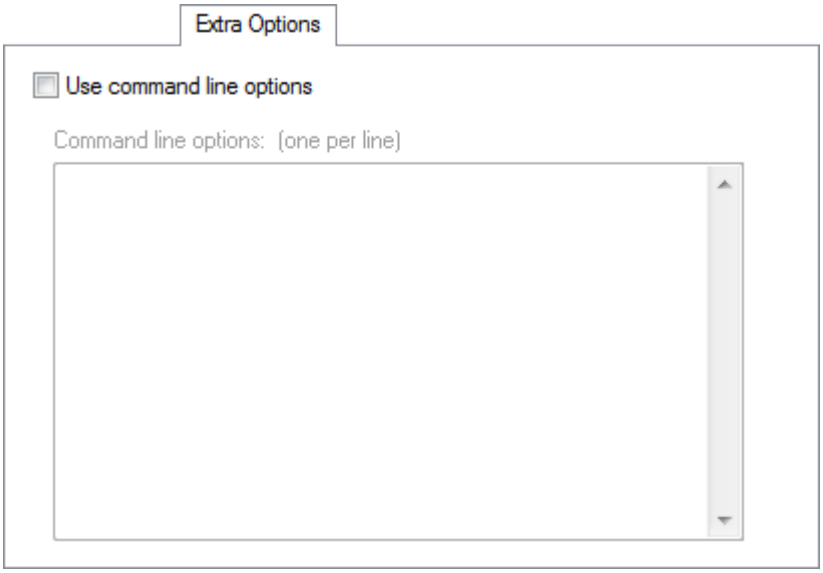
Library file Specify explicitly the path to the library file (`.dll`) used for SDM debug authentication.

Apply authentication if status cannot be determined

Use this option if, for some reason, the detection mechanism fails and you want to force using the authentication procedure. This could be the case if you are using a device that is not explicitly supported yet by IAR Embedded Workbench.

Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.



Use command line options

Specify command line arguments that are not supported by the IDE to be passed to C-SPY.

Note that it is possible to use the `/args` option to pass command line arguments to the debugged application.

The syntax is:

```
/args arg0 arg1 ...
```

Multiple lines with `/args` are allowed, for example:

```
/args --logfile log.txt

/args --verbose
```

If you use `/args`, these variables must be defined in your application:

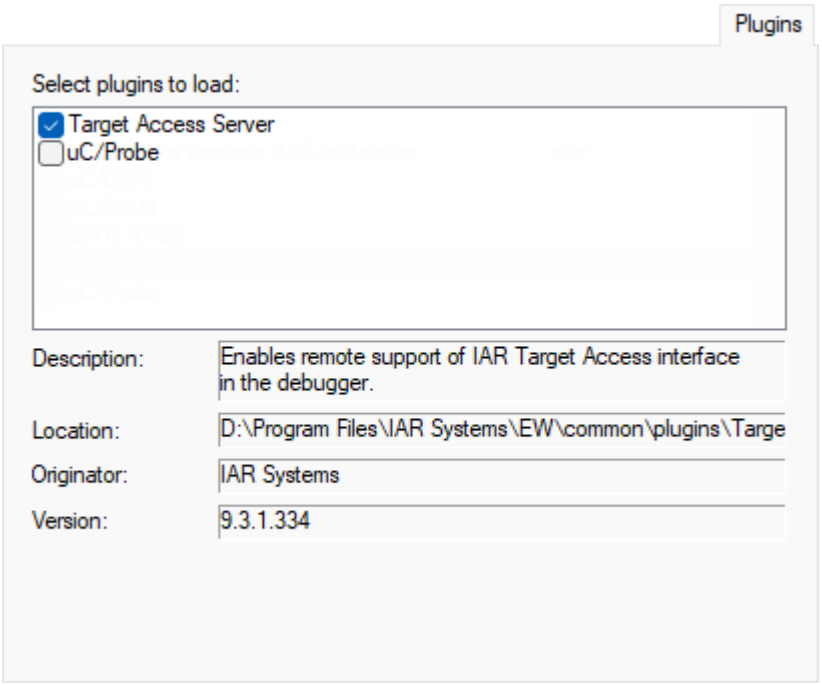
```
/* __argc, the number of arguments in __argv. */
__no_init __root int __argc;

/* __argv, an array of pointers to strings that holds the arguments; must
   be large enough to fit the number of parameters.*/
__no_init __root const char * __argv[MAX_ARGS];

/* __argvbuf, a storage area for __argv; must be large enough to hold all
   command line parameters. */
__no_init __root char __argvbuf[MAX_ARG_SIZE];
```

Plugins

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



Select plugins to load

Selects the plugin modules to be loaded and made available during debug sessions. The list contains the plugin modules delivered with the product installation.

Description

Describes the plugin module.

Location

Informs about the location of the plugin module.

Generic plugin modules are stored in the `common\plugins` directory. Target-specific plugin modules are stored in the `arm\plugins` directory.

Originator

Informs about the originator of the plugin module, which can be modules provided by IAR or by third-party vendors.

Version

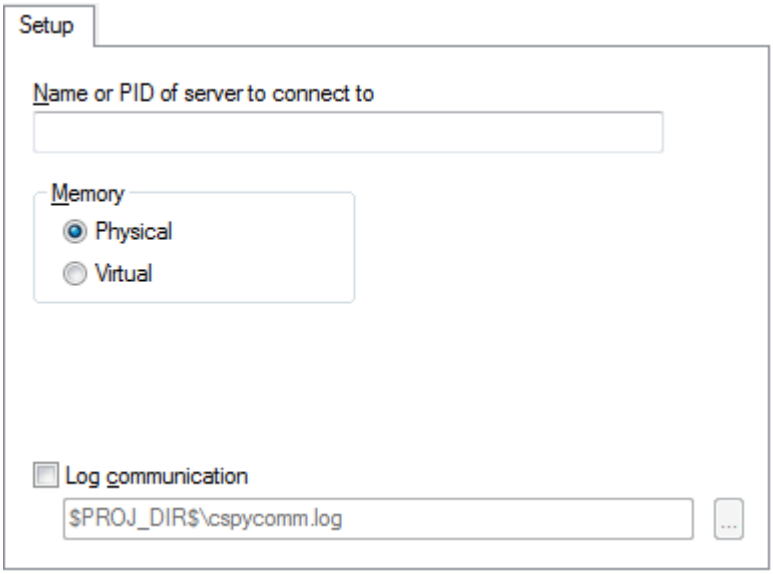
Informs about the version number.

REFERENCE INFORMATION ON C-SPY HARDWARE DEBUGGER DRIVER OPTIONS

For reference information about the **Breakpoints** options page, see *Breakpoints options, page 112*.

CADI — Setup

The **Setup** options control the C-SPY CADI driver:



Name or PID of server to connect to

To use the C-SPY CADI driver, you must start an Arm Fast model virtual platform with a CADI server running (using the `-S` command when starting the Fast model). If multiple Fast model platforms are running, use this option to specify the name or the Process ID of the Fast model server to connect to.

Memory

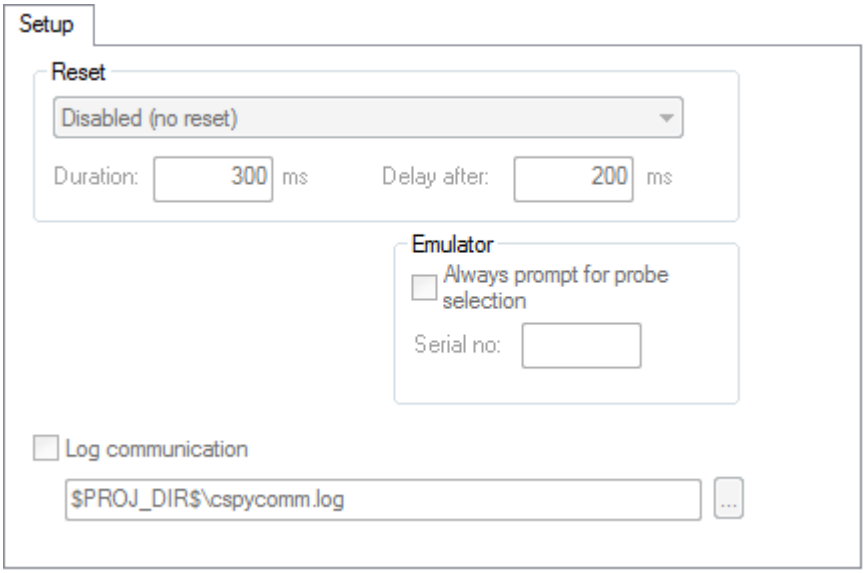
The memory type of the Fast model platform. Choose between **Physical** and **Virtual**.

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.

CMSIS-DAP — Setup

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 only applicable 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 `nSRST`/`nRESET` 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 `VECTRESET` bit—the peripheral units are not affected.
System	Resets the core and peripherals. This is the default reset strategy.
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	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. A watchdog timer might be disabled. Special debug modes, such as debugging in power-saving modes, might be turned on. This option is only available for some devices.
Reset by watchdog or reset register	Resets the processor using a software reset register or a watchdog reset. Peripheral units might not be reset. This reset strategy is recommended when the processor cannot be stopped at the reset vector using the hardware reset. Device-specific software reset. This option is only available for some devices.
Reset and halt after bootloader	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. Depending on the device, this reset strategy is implemented using the hardware, core, or system reset. This option is only available for some devices.
Reset and halt before bootloader	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. This option is only available for some devices.

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 40*, and *Debugging code in RAM, page 41*.



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 38*.

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, and thus inaccessible for the debugger.

This option applies to the hardware reset, and to those custom reset styles that use the hardware reset.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

Emulator

These options are used for identifying the debug probe to use.

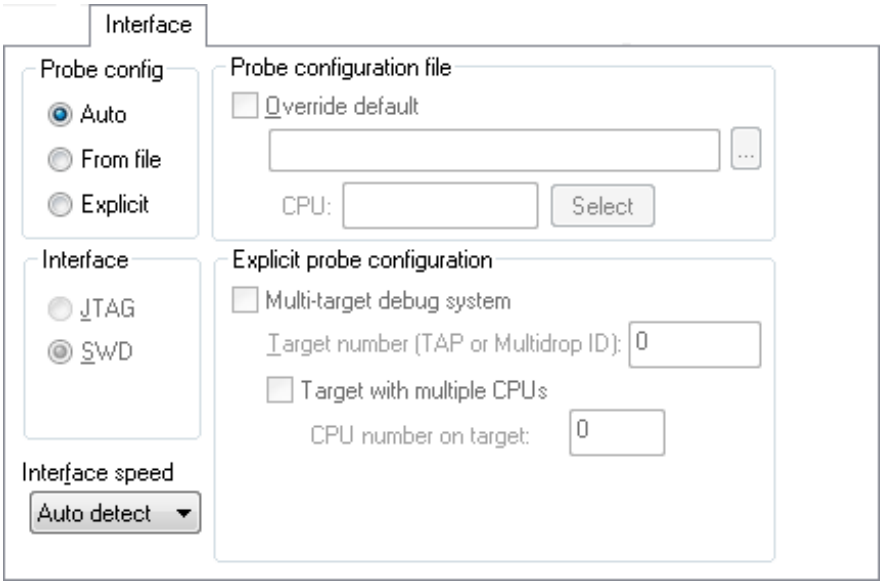
Always prompt for probe selection	Makes C-SPY always ask you to confirm which probe to use, if more than one debug probe is connected to the host computer.
Serial no	Enter the serial number of the debug probe you are using.

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.

CMSIS-DAP — Interface

The **Interface** 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.

Interface speed

Specifies 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**.

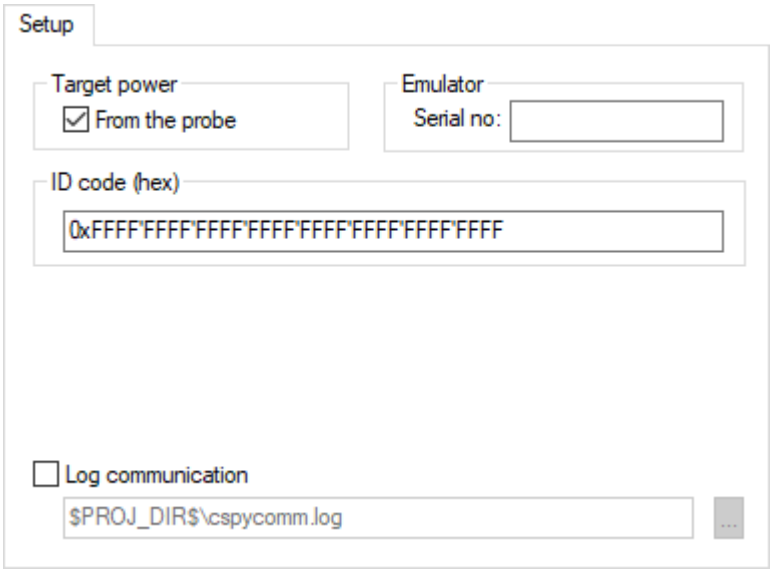
Target with multiple CPUs
Specifies that the target has several CPUs.

CPU number on target
If the debug system is a multicore SWD, specify the CPU number on the DAP.

If the debug system is a JTAG scan chain and there are several CPUs at the TAP position, specify the CPU number on the TAP.

E2 — Setup

The **Setup** options control the E2/E2Lite interface.



Target power

Select **From the probe** to specify that the supply voltage to your target device will be supplied by the probe.

Emulator

If more than one debug probe is connected to the host computer, use the **Serial no** option to select the correct one.

ID code (hex)

The hexadecimal ID code protects certain devices with an authentication check before the debug session is started. If an incorrect ID code is entered, a connection error will occur.

For more information, see the hardware manual for your device.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

GDB Server

The **GDB Server** options control the C-SPY GDB Server for the STR9-comStick evaluation board.



TCP/IP address or hostname

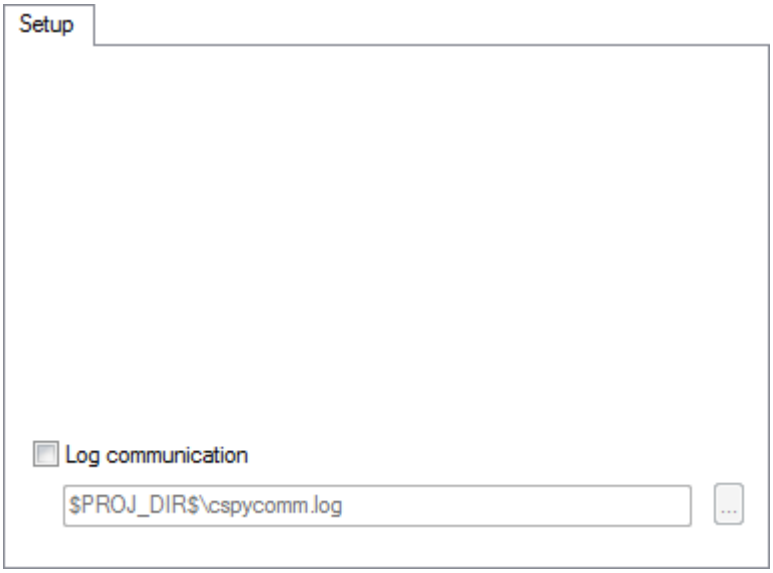
Specify the IP address and port number of a GDB server—by default the port number `3333` is used. The TCP/IP connection is used for connecting to a J-Link server running on a remote computer.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

G+LINK — Setup

The **Setup** options specify the G+LINK interface.

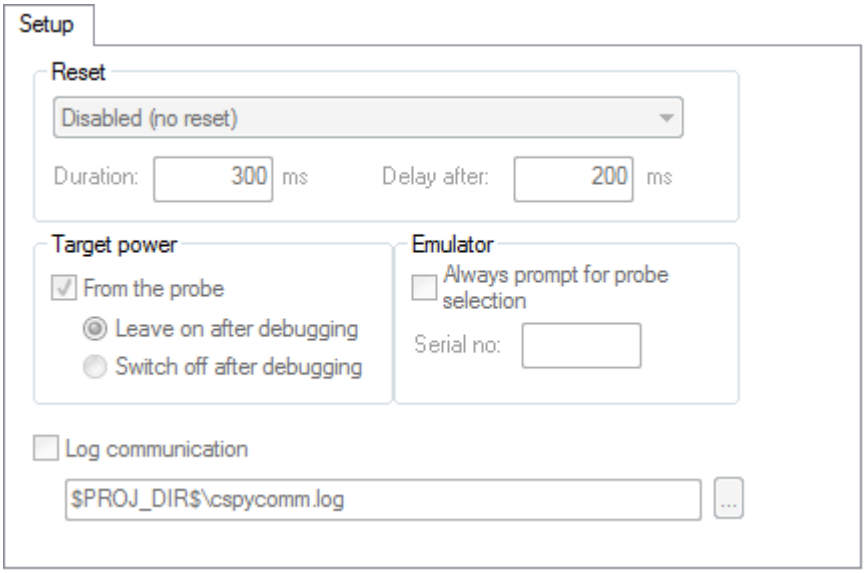


Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

I-jet — Setup

The **Setup** options control the I-jet and I-jet Trace in-circuit debugging probes.



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 `PC` to the program entry address and `SP` to the initial stack pointer value. This is a software reset.
Hardware	The probe toggles the `nSRST`/`nRESET` line on the JTAG connector to reset the device. 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 `VECTRESET` bit—the peripheral units are not affected. For Cortex-M devices only.
System	Resets the core and peripheral units by setting the `SYSRESETREQ` bit in the `AIRCR` 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 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	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. A watchdog timer might be disabled. Special debug modes, such as debugging in power-saving modes, might be turned on. This option is only available for some devices.
Reset by watchdog or reset register	Resets the processor using a software reset register or a watchdog reset. Peripheral units might not be reset. This reset strategy is recommended when the processor cannot be stopped at the reset vector using the hardware reset. Device-specific software reset. This option is only available for some devices.
Reset and halt after bootloader	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. Depending on the device, this reset strategy is implemented using the hardware, core, or system reset. This option is only available for some devices.

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 40*, and *Debugging code in RAM, page 41*.



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 38*.

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, and thus inaccessible for the debugger.

This option applies to the hardware reset, and to those custom reset styles that use the hardware reset.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

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.

Emulator

These options are used for identifying the debug probe to use.

Always prompt for probe selection Makes C-SPY always ask you to confirm which probe to use, if more than one debug probe is connected to the host computer.

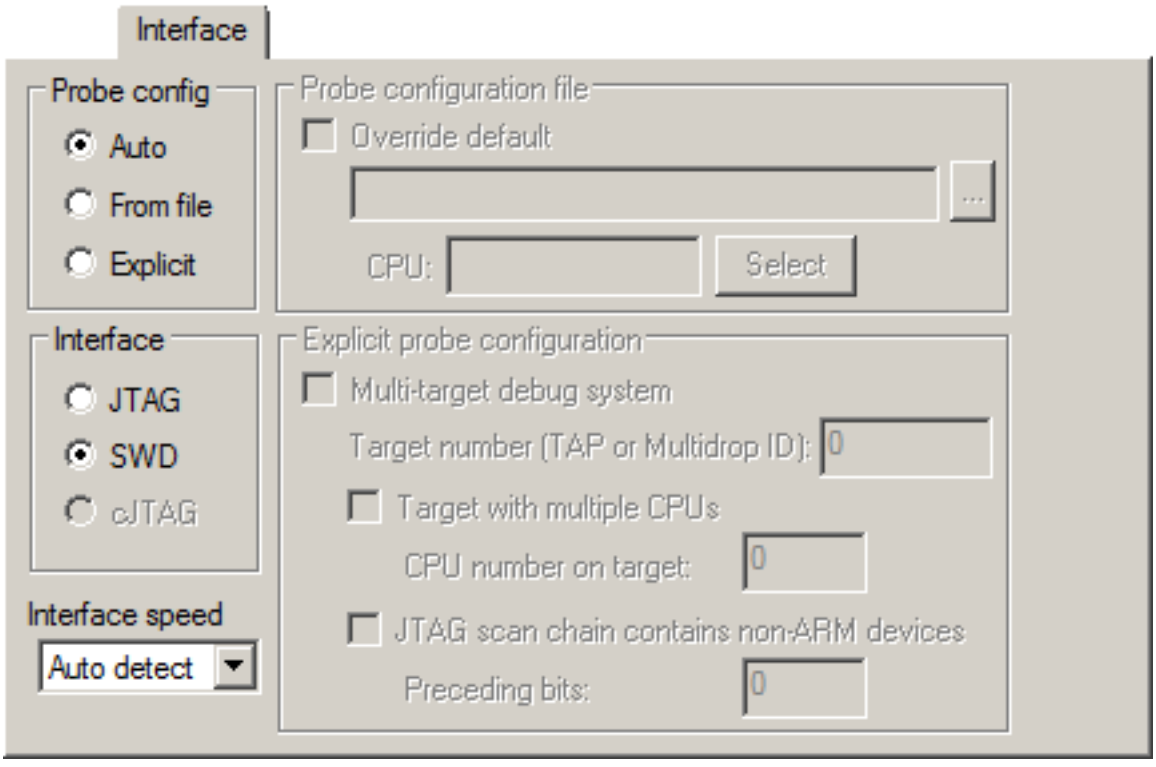
Serial no Enter the serial number of the debug probe you are using.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required.

I-jet — Interface

The **Interface** options specify the interface between I-jet or I-jet Trace and the target system.



Probe config

Auto The I-jet driver automatically identifies the target CPU. It uses the default probe configuration file, if there is one. This works best if there is only one CPU present.

From file Specifies that the probe configuration file needs to be overridden, or that there are several target CPUs.

Explicit Specify how to find the target CPU.

Interface

Selects the communication interface between the debug probe and the target system. Choose between:

JTAG	Uses the JTAG interface.
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 165*.
cJTAG	Uses the cJTAG interface.

Interface speed

Specifies 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**	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 and are counted from the TDO end (TAP number `0` is the one connected to the TDO line). If there are several CPUs at the TAP position, you also need to specify the **CPU number on target**.

Target with multiple CPUs
Specifies that the target has several CPUs.

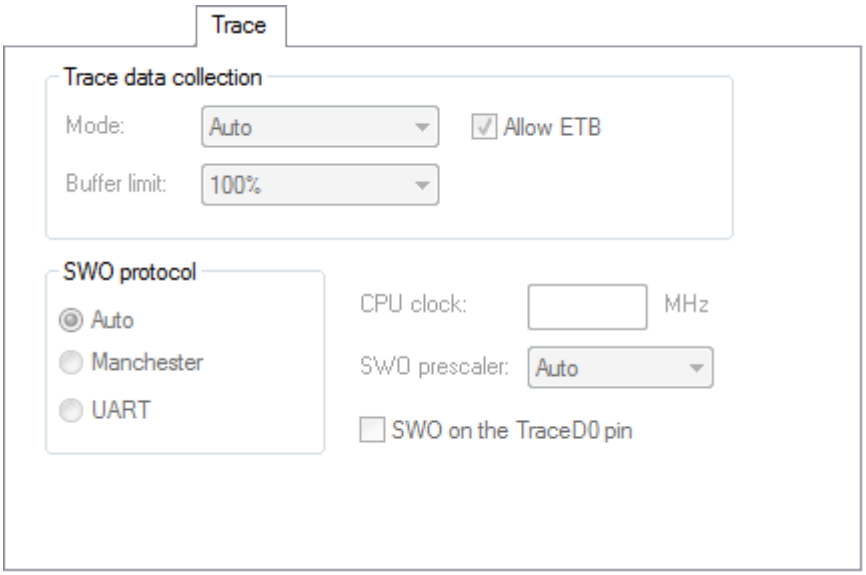
CPU number on target
If the debug system is a multicore SWD, specify the CPU number on the DAP.
If the debug system is a JTAG scan chain and there are several CPUs at the TAP position, specify the CPU number on the TAP.

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 number of IR bits preceding the Arm core to connect to.

I-jet — Trace

The **Trace** options specify the trace behavior for I-jet.



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.

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

Limits probe memory use. This option is only available if parallel (ETM) mode is used, either explicitly through **Parallel (ETM)** or implicitly through **Auto**.

Collected trace data is stored in probe memory.

Because reading and decoding large amounts of trace data takes 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). Using the **Buffer limit** option limits the buffer use to a percentage of the total memory size. Using more probe memory yields more trace data, but will take longer to see results and use up more host memory. C-SPY will retrieve the most recent samples from the trace probe, and the rest of the collected trace data will be discarded.

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 a 4-bit sample), so you must use your own judgment to see what buffer limit that provides a good balance between the size of decoded data and C-SPY performance.

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 and for calculating timestamps.

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 *SWO Configuration dialog box, page 167*.

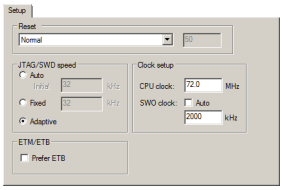
SWO on the TraceD0 pin

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.

J-Link/J-Trace — Setup

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 only available for STM32 devices.

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 only available for LPC11xx and LPC13xx devices.

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) After reset, J-Link continuously tries to halt the CPU using a breakpoint. Typically, this halts the CPU shortly after reset—in most systems, the CPU can execute some instructions before it is halted.

	This is a hardware reset.
Hardware, halt at 0 (4)	Halts the processor by placing a breakpoint at the address zero. Note that this is not supported by all Arm microcontrollers. This is a hardware reset.
Hardware, halt using DBGRQ (5)	After reset, J-Link continuously tries to halt the CPU using DBGRQ. Typically, this halts the CPU shortly after reset—in most systems, the CPU can execute some instructions before it is halted. This is a hardware reset.
Software (-)	Sets `PC` to the program entry address. This is a software reset.
Software, Analog devices (2)	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. This is a software reset.
Hardware, NXP LPC (9)	This strategy is only available if you have selected such a device from the **Device** drop-down list on the **General Options>Target** page. This is a hardware reset specific to NXP LPC devices.
Hardware, Atmel AT91SAM7 (8)	This strategy is only available if you have selected such a device from the **Device** drop-down list on the **General Options>Target** page. This is a hardware reset specific for the **Atmel AT91SAM7** family.

For more information about different reset strategies, see the J-Link/J-Trace documentation available at *www.segger.com* .

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 40*, and *Debugging code in RAM, page 41*.



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 38*.

JTAG/SWD speed

Specify the JTAG communication speed in kHz. Choose between:

Auto
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—1000 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. If the CPU starts at a low clock speed, you might need to set a lower initial value, for example 32 kHz.

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.

The initial value must be in the range 1–50000 kHz.

Fixed
Sets the JTAG communication speed in kHz. The value must be in the range 1–50000 kHz.

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.

Adaptive
Synchronizes the clock to the processor clock outside the core. Works only with Arm devices that have the `RTCK` JTAG signal available. For information about adaptive speed, see the J-Link/J-Trace documentation available at *www.segger.com* .

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 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 167*.

ETM/ETB

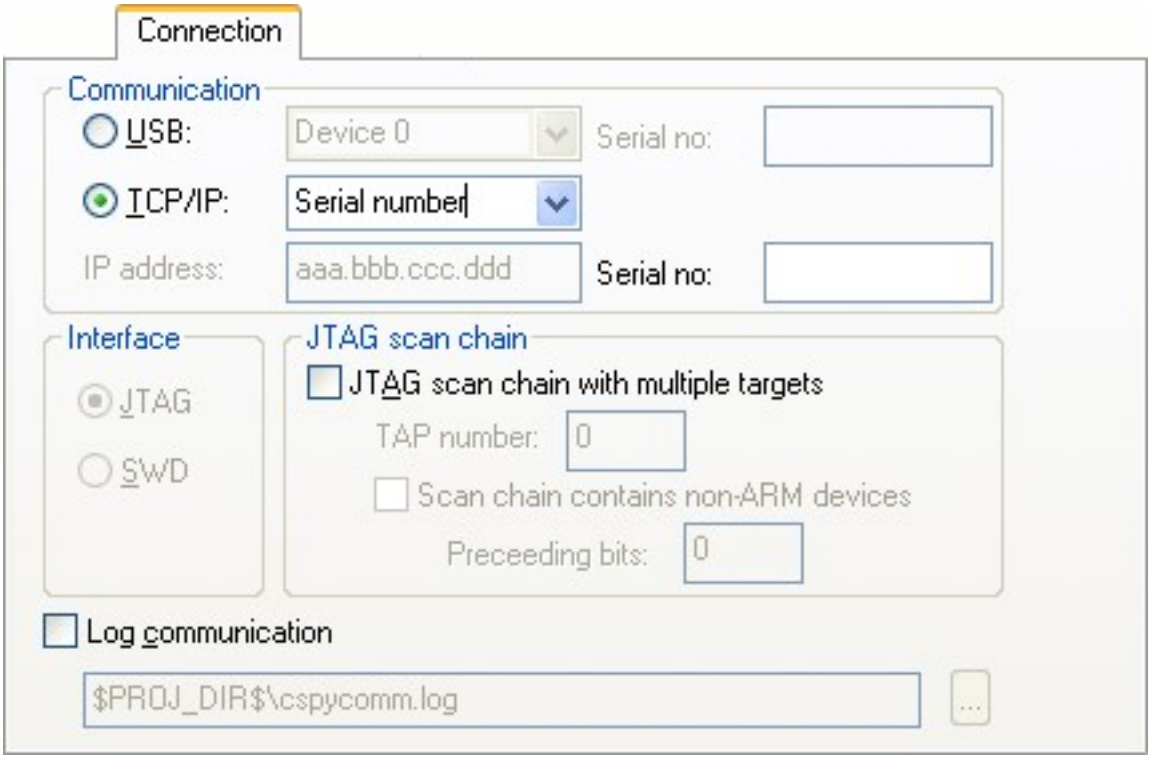
The **Prefer ETB** option selects ETB trace instead of ETM trace, which is the default.



This option only applies to J-Trace.

J-Link/J-Trace — Connection

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 165*.

JTAG scan chain

Specifies the JTAG scan chain. Choose between:

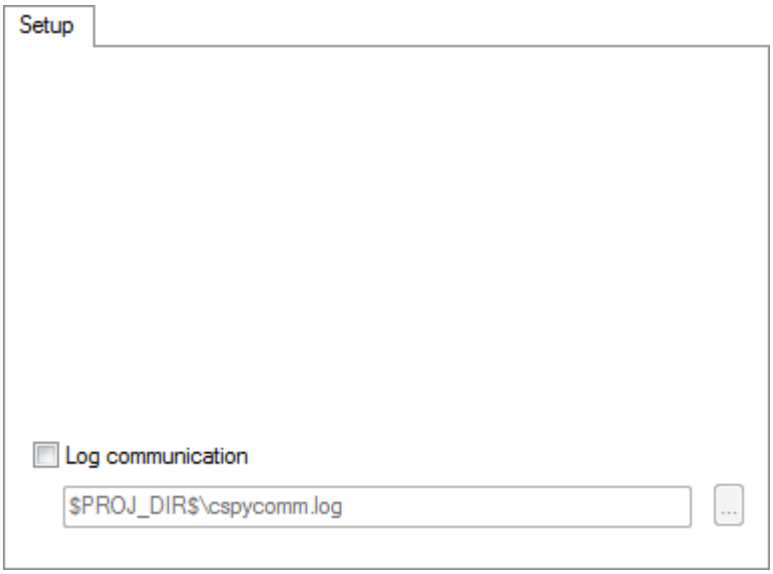
JTAG scan chain with multiple targets	Specifies that there is more than one device on the JTAG scan chain.
TAP number	Specify the TAP (Test Access Port) position of the device you want to connect to. The TAP numbers start from zero.
Scan chain contains non-Arm devices	Enables JTAG scan chains that mix Arm devices with other devices like, for example, FPGA.
Preceeding 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.

Nu-Link — Setup

The **Setup** options specify the Nu-Link interface.



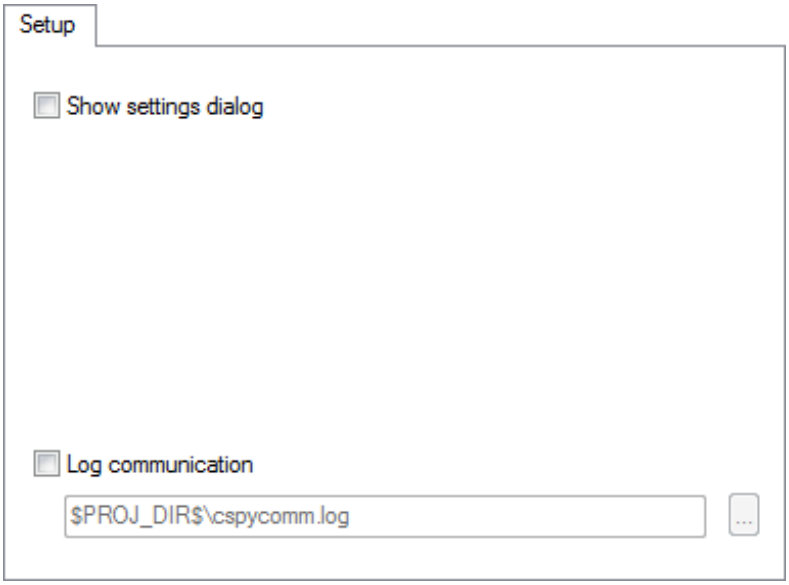
For information about this driver, see the file `NuMicro Cortex-M IAR ICE driver user manual.pdf` available in the `arm\bin\Nu-Link` directory.

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.

PE micro — Setup

The **Setup** options specify the PE micro interface.



Show settings dialog

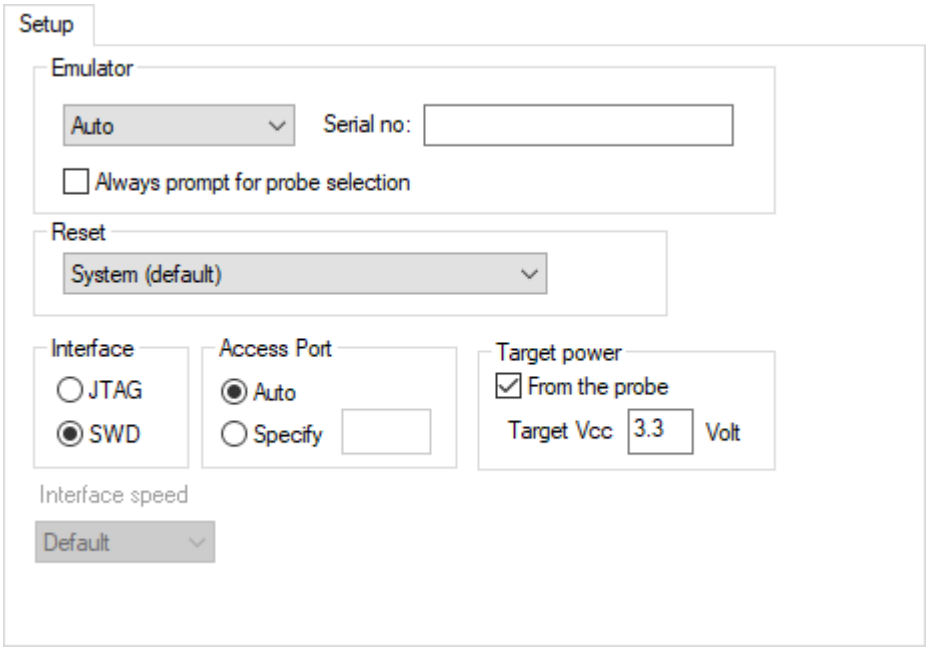
Displays the **P&E Connection Manager** dialog box. For more information, see the file `PEMicroSettings.pdf`, available in the `arm\doc` directory.

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.

ST-LINK — Setup

The **Setup** options specify the ST-LINK interface.



Emulator

Specify the emulator you are using. Choose between **Auto** (default) where the ST-LINK driver automatically selects the first emulator, or **ST-LINK/V1**, **ST-LINK/V2** or **ST-LINK/V3**.

Serial no — If more than one debug probe is connected to the host computer, enter the serial number of the debug probe to identify the correct one.

Always prompt for probe selection — Makes C-SPY always ask you to confirm which probe to use, if more than one debug probe is connected to the host computer.

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:

System (0, default)	Resets the core and peripheral units by setting the `SYSRESETREQ` bit in the `AIRCR` register. Reset vector catch is used for stopping the CPU at the reset vector before the first instruction is executed. Only available for Cortex-M devices.
Hardware (1)	The probe toggles the `nSRST/nRESET` line on the JTAG connector to reset the device. This reset usually also resets the peripheral units. 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 and when debugging Cortex-M devices.
Core (3)	Resets the core via the `VECTRESET` (Cortex-M) or `CWRR` (Cortex-A) bit—the peripheral units are not affected.
Software reset (4)	Sets the `PC` to program entry address and `SP` to the initial stack pointer value.

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.

Access port

Use this option to specify the access port when using devices with more than one core—shared mode debugging. Choose between:

Auto (default)	A predefined value for the device is used. For most devices, the access port defaults to 0.
Specify	For devices with more than one core, you can specify the number of the access port on the device you want to connect to, `0-3`.

Target power

Select **From the probe** to specify that the supply voltage to your target device will be supplied by the probe. Use **Target Vcc** to specify the voltage.

Interfacespeed

Specify the JTAG and SWD communication speed. Choose between:

Default — Automatically uses the highest possible frequency for reliable operation. This is the only option for ST-LINK version 1.

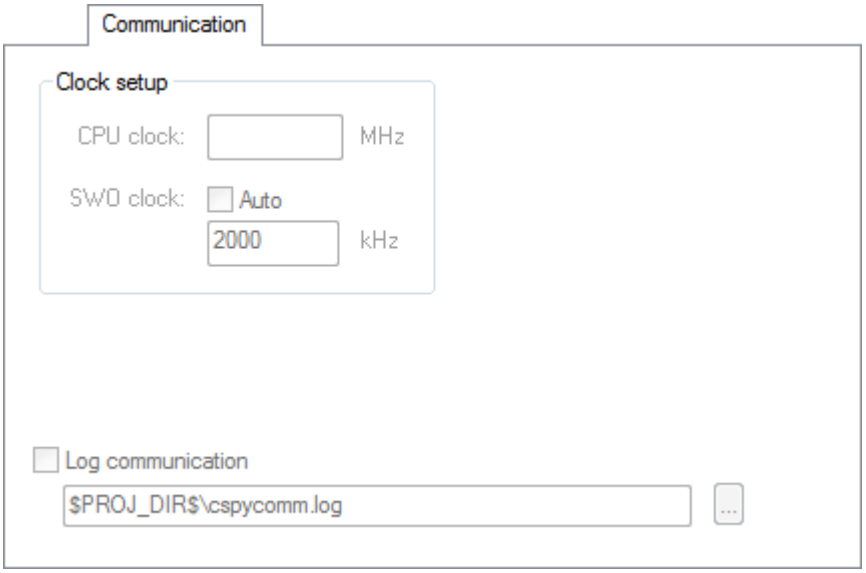
***n* MHz** — For ST-LINK version 2 and ST-LINK version 3, sets the JTAG and SWD communication speed to the selected frequency. This option is not available for ST-LINK version 1.

If the selected frequency is not supported by the specified probe, the closest and lowest supported frequency is automatically selected.

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.

ST-LINK — Communication

The **Communication** options specify the ST-LINK 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 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 167*.

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.

ST-LINK — Multicore

The **Multicore** options enable multicore and shared mode debugging for the ST-LINK interface.



Enable multicore debugging/shared mode

Enables multicore and shared mode debugging where two or more instances of IAR Embedded Workbench or an external tool share the same debug probe. Each instance accesses one core on the target device.

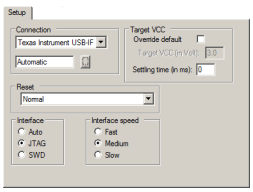
Multicore and shared mode debugging requires that you have installed the ST-LINK server. To install the server, run `arm\drivers\ST-LINK\st-stlink-server.msi`.

See also, the option **Access Port** at *ST-LINK — Setup, page 497*.

For more information, see the application note from ST Microelectronics, *STM32H7x5/x7 dual-core microcontroller debugging* at **st.com**.

TI MSP-FET — Setup

The **Setup** options specify the TI MSP-FET interface.



Connection

Controls the communication between C-SPY and the target device.

The C-SPY TI MSP-FET debugger can communicate with the target device via a number of different debug probes. Select the probe you are using. If you select `Automatic`, the debugger will automatically connect to the correct port.

Some emulator drivers support multiple emulators connected to the same host computer. Each emulator requires its own instance of IAR Embedded Workbench and each instance must identify its emulator. To identify an emulator, click the browse button to display a list of all detected emulators. To identify a connection, click the port in the list and the Mode LED on the attached emulator will light up.

Target VCC

Specify the voltage provided by the USB interface:

Override default	Overrides the default voltage. The default voltage is displayed in the **Target VCC** text box, when you deselect the **Override default** option.
Target VCC	Specify the voltage with one decimal's precision in the range 1.0-4.0 V. This option can only be used when your target device is powered by the debug probe.
Settling time	Specify a delay (in milliseconds) that will be used between switching on the target VCC and starting the identification of the Arm device.



User-defined millisecond timeouts and delays are generated using the host operating system clock ticks. Because the host operating system is not a real-time OS, the actual delay or timeout will vary each time with approximately 5–20 ms.

Reset

Selects the reset strategy to be used when the debugger starts.

Normal (0)	This is the default strategy. It does whatever is the best way to reset the target device.

Interface

Determines the debug interface to use:

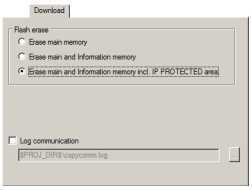
Auto	Selects the debug interface automatically.
JTAG	Selects the JTAG interface.
SWD	Selects the SWD interface. SWD uses fewer pins than JTAG.

Interface speed

Sets the JTAG communication speed. Choose between **Fast**, **Medium**, and **Slow**.

TI MSP-FET — Download

The **Download** options specify the TI MSP-FET interface.



Erase main memory

Erases only the main flash memory before download. The Information memory is not erased.

Erase main and Information memory

Erases both flash memories—main and Information memory—before download.

Erase main and Information memory inc. IP PROTECTED area

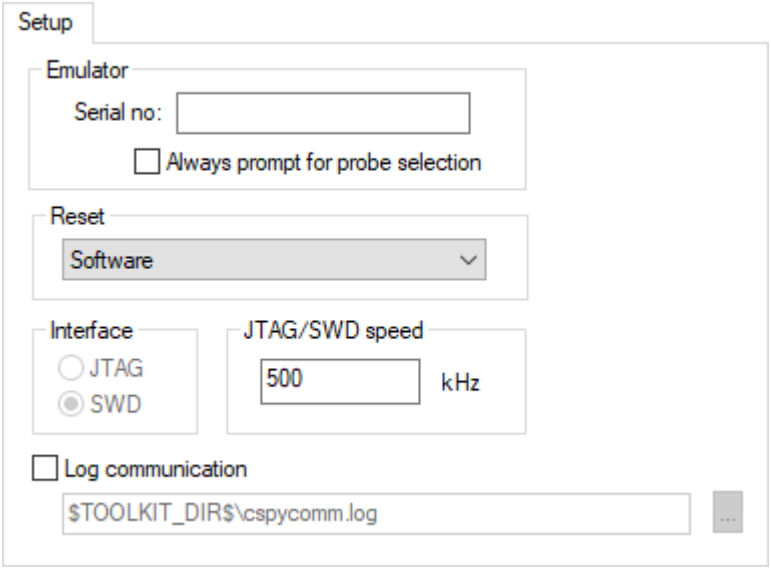
Erases the main and Information flash memories, including the IP protected area before download.

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.

TI Stellaris — Setup

The **Setup** options specify the TI Stellaris interface.



Emulator

These options are used for identifying the debug probe to use.

Always prompt for probe selection Makes C-SPY always ask you to confirm which probe to use, if more than one debug probe is connected to the host computer.

Serial no Enter the serial number of the debug probe you are using.

Reset

Selects the reset strategy to be used when the debugger starts. Based on your hardware, one of the strategies is the default. Choose between:

Software (core) (0) Resets the core. Sets `PC` to the program entry address and `SP` to the initial stack pointer value.

System (1) Resets the core and peripheral units. Sets `PC` to the program entry address and `SP` to the initial stack pointer value.

Interface

Selects the communication interface between the TI Stellaris 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 165*.

JTAG/SWD speed

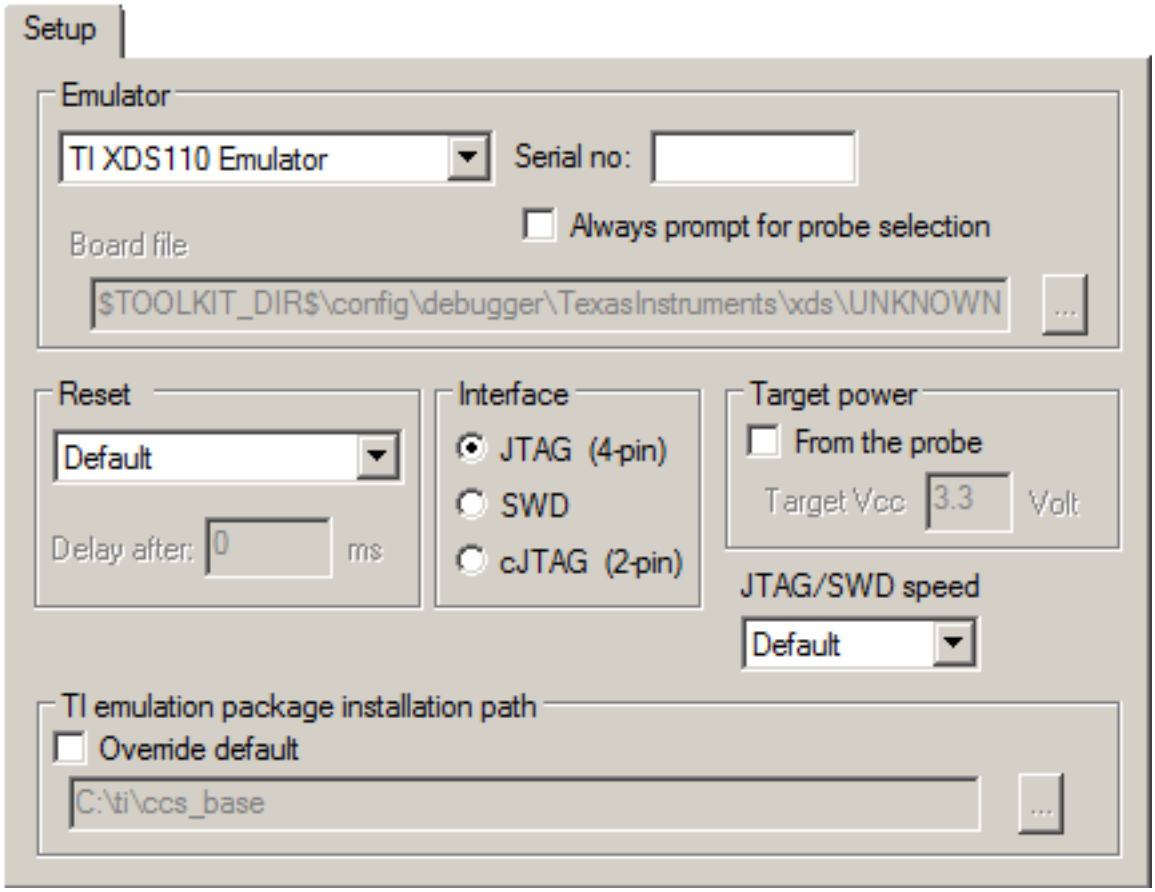
Specify the JTAG communication speed in kHz.

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.

TI XDS — Setup

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 the **Serial no** option to select the correct one.

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**.

To override the default board file, first select the menu command **Specify custom board file** in the **Emulator** drop-down menu and then specify a board file using the **Board file** option.

Reset

Selects the reset strategy to be used when the debugger starts. Based on your hardware, one of the strategies is the default. Choose between:

For Cortex-M devices:

CPU (0)	Resets the CPU.
System (1)	Resets the core and peripheral units.
Board (2)	Uses the reset pin to reset the device. Only available for CC26xx and CC13xx devices.
Board (3)	Uses the reset pin to reset the device. Run and halt with delay. Available for all other Cortex-M devices.

For non-Cortex-M devices:

Software (0)	Sets `PC` to the program entry address.
Hardware (1)	Uses the reset pin to reset the device.

Interface

Select the communication interface between the XDS debug probe and the target system.

Target power

Select **From the probe** to specify that the supply voltage to your target device will be supplied by the probe. Use **Target Vcc** to specify the voltage.

This option is only available for the stand-alone TI XDS110 debug probe.

JTAG/SWD speed

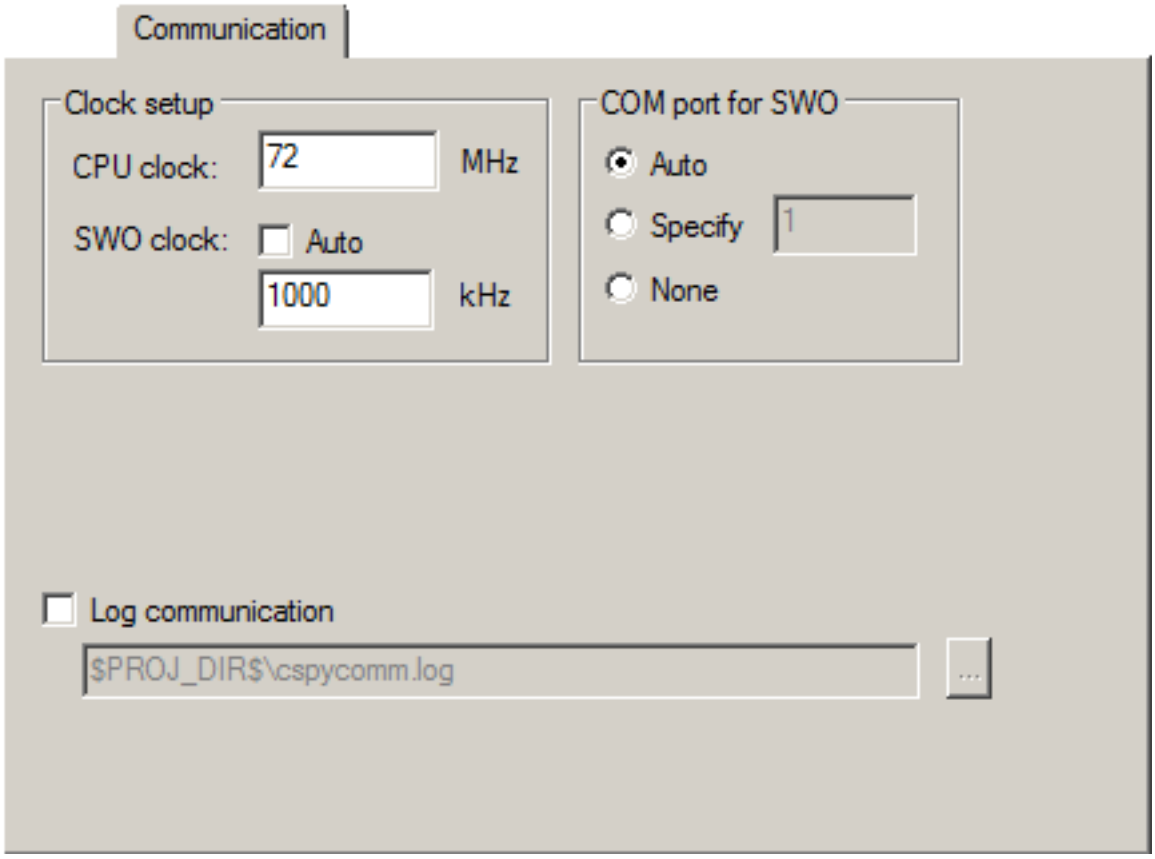
Specify the JTAG communication speed.

TI emulation package installation path

Select **Override default** to override the default installation path of the Texas Instruments emulation package.

TI XDS — Communication

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 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 167*.

COM port for SWO

Specifies the COM port to use for the SWO communication. Choose between:

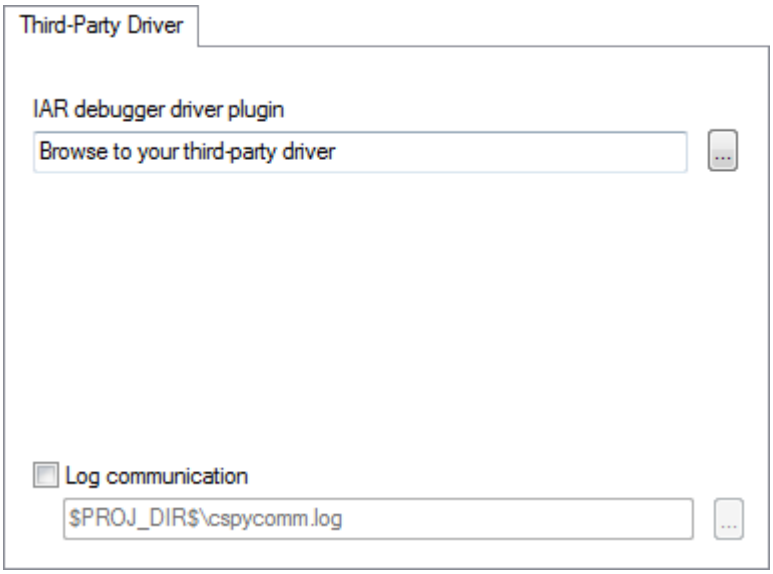
Auto	The debugger automatically identifies the COM port that is associated with the debug probe.
Specify	Specify the number of the COM port, 1-256.
None	The debugger leaves the COM port unused and SWO trace is disabled.

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

Contents

Reference information on C-SPY driver menus ... 509
C-SPY driver ... 510
Simulator menu (IAR native) ... 510
Simulator menu (Imperas) ... 512
Reference information on the C-SPY simulator ... 512
Simulated Frequency dialog box ... 512
Reference information on the C-SPY hardware debugger drivers ... 513
CADI menu ... 513
CMSIS-DAP menu ... 513
E2 menu ... 515
GDB Server menu ... 516
G+LINK menu ... 517
I-jet menu ... 517
J-Link menu ... 520
Nu-Link menu ... 523
ST-Link menu ... 524
TI MSP-FET menu ... 526
TI Stellaris menu ... 527
TI XDS menu ... 528
Resolving problems ... 531
No contact with the target hardware ... 531

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 510*
- *Simulator menu (IAR native), page 510*
- *Simulator menu (Imperas), page 512*
- *CADI menu, page 513*
- *CMSIS-DAP menu, page 513*
- *E2 menu, page 515*
- *GDB Server menu, page 516*
- *G+LINK menu, page 517*
- *I-jet menu, page 517*
- *J-Link menu, page 520*
- *Nu-Link menu, page 523*
- *ST-Link menu, page 524*
- *TI MSP-FET menu, page 526*

- *TI Stellaris menu, page 527*
- *TI XDS menu, page 528*

C-SPY driver

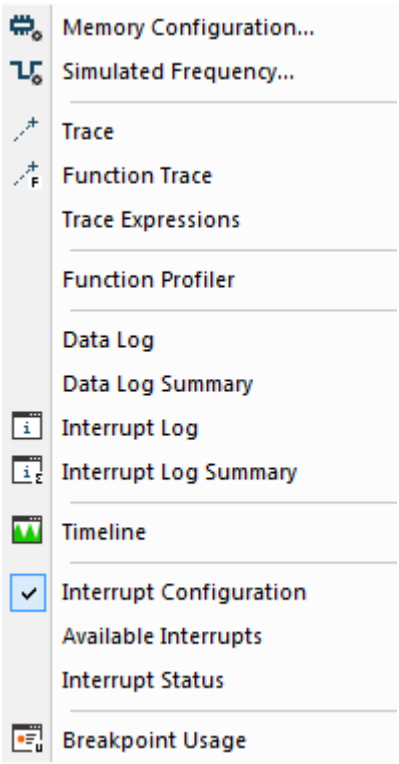
Before you start the C-SPY debugger, you must first specify a C-SPY driver in the **Options** dialog box, using the option **Debugger>Setup>Driver**.

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 documentation write "choose ***C-SPY driver>***" (or just ***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 (IAR native)

The C-SPY simulator uses one of two debugger drivers. Which driver depends on which core or device you have selected for your application project, see *The IAR C-SPY Simulator, page 28*. When you use the IAR native simulator driver, this **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 146*.

Trace



Opens a window which displays the collected trace data, see *Trace window, page 170*.

Function Trace



Opens a window which displays the trace data for function calls and function returns, see*Function Trace window, page 178*.

Function Profiler Opens a window which shows timing information for the functions, see *Function Profiler window, page 229*.

Data Log Opens a window which logs accesses to up to four different memory locations or areas, see *Data Log window, page 201*.

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 203*.

Interrupt Log



Opens a window which displays the status of all defined interrupts, see *Interrupt Log window, page 318*.

Interrupt Log Summary



Opens a window which displays a summary of the status of all defined interrupts, see *Interrupt Log Summary window, page 321*.

Timeline



Opens a window which gives a graphical view of various kinds of information on a timeline, see *The application timeline, page 195*.

Simulated Frequency



Opens the **Simulated Frequency** dialog box where you can specify the simulator frequency used when the simulator displays time information, for example in the log windows. Note that this does not affect the speed of the simulator. For more information, see *Simulated Frequency dialog box, page 512*.

Interrupt Configuration Opens a window where you can configure C-SPY interrupt simulation, see *Interrupt Configuration window, page 312*.

Available Interrupts Opens a window with an overview of all available interrupts. You can also force an interrupt instantly from this window, see *Available Interrupts window, page 314*.

Interrupt Status Opens a window from where you can instantly trigger an interrupt, see *Interrupt Status window, page 316*.

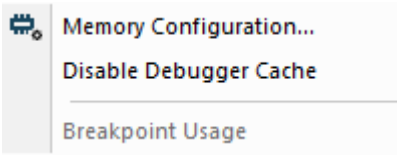
Breakpoint Usage



Displays a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

Simulator menu (Imperas)

The C-SPY simulator uses one of two debugger drivers. Which driver depends on which core or device you have selected for your application project, see *The IAR C-SPY Simulator, page 28*. When you use the Imperas simulator driver, this **Simulator** menu is added to the menu bar:



Menu commands

These commands are available on the menu:

Memory Configuration Displays a dialog box, see *Memory Configuration dialog box for the C-SPY simulator, page 146*.

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.

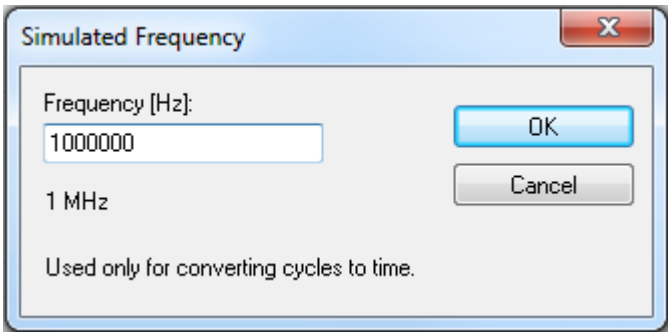
Breakpoint Usage Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

REFERENCE INFORMATION ON THE C-SPY SIMULATOR

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

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. Not available for all cores and devices.

Frequency

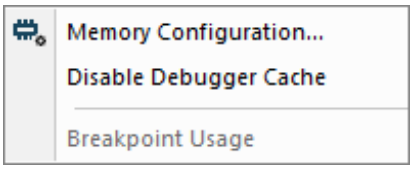
Specify the frequency in Hz.

REFERENCE INFORMATION ON THE C-SPY HARDWARE DEBUGGER DRIVERS

This section gives additional reference information on the C-SPY hardware debugger drivers, reference information not provided elsewhere in this documentation.

CADI menu

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



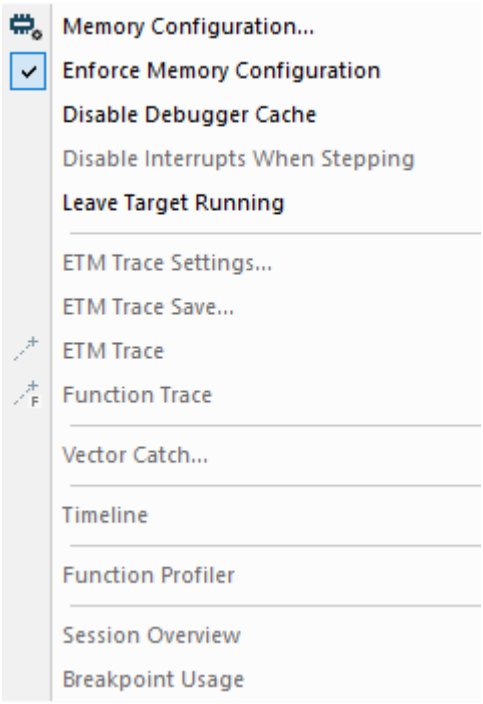
Menu commands

These commands are available on the menu:

Memory Configuration	Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.
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.
Breakpoint Usage	Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

CMSIS-DAP menu

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



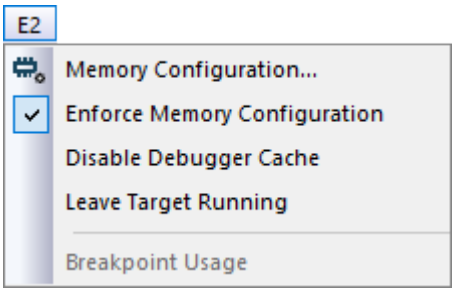
Menu commands

These commands are available on the menu:

Memory Configuration	Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.
Enforce Memory Configuration	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
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 statements 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.
Leave Target Running	Leaves the application running on the target hardware after the debug session is closed. Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.
ETM Trace Settings	Displays a dialog box, see *ETM Trace Settings dialog box (I-jet), page 161*.
ETM Trace Save	Displays a dialog box, see *Trace Save dialog box, page 193*.
ETM Trace	Opens the ETM Trace window, see *Trace window, page 170*.
Function Trace	Opens a window, see *Function Trace window, page 178*.
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 115*. Note that this command is not available for all Arm cores.
Timeline	Opens a window, see *The application timeline, page 195*. 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 229*.
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 101*.

E2 menu

When you are using the C-SPY E2/E2Lite driver, the **E2** menu is added to the menu bar.



Menu commands

These commands are available on the menu:

Memory Configuration Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.

Enforce Memory Configuration Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.

Disable Debugger Cache Disables memory caching and memory range checking in C-SPY.

Normally, C-SPY uses the memory range information in the **Memory Configuration** dialog box both to restrict access to certain parts of target memory and to cache target memory contents for improved C-SPY performance. Under certain rare circumstances, this is not appropriate, and you can choose **Disable Debugger Cache** to turn off the caching and memory range checking completely. All accesses from C-SPY will then result in corresponding accesses to the target system. Some of those circumstances are:

- When memory is remapped at runtime and cannot be specified as a fixed set of ranges.
- When the memory range setup is incorrect or incomplete.

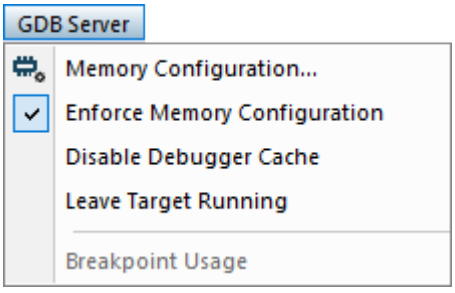
Leave Target Running Leaves the application running on the target hardware after the debug session is closed.

Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.

Breakpoint Usage Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

GDB Server menu

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



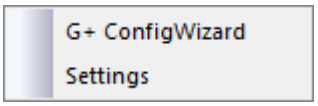
Menu commands

These commands are available on the menu:

Memory Configuration	Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.
Enforce Memory Configuration	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
Disable Debugger Cache	Disables memory caching and memory range checking in C-SPY. Normally, C-SPY uses the memory range information in the **Memory Configuration** dialog box both to restrict access to certain parts of target memory and to cache target memory contents for improved C-SPY performance. Under certain rare circumstances, this is not appropriate, and you can choose **Disable Debugger Cache** to turn off the caching and memory range checking completely. All accesses from C-SPY will then result in corresponding accesses to the target system. Some of those circumstances are: • When memory is remapped at runtime and cannot be specified as a fixed set of ranges. • When the memory range setup is incorrect or incomplete.
Leave Target Running	Leaves the application running on the target hardware after the debug session is closed. Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.
Breakpoint Usage	Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

G+LINK menu

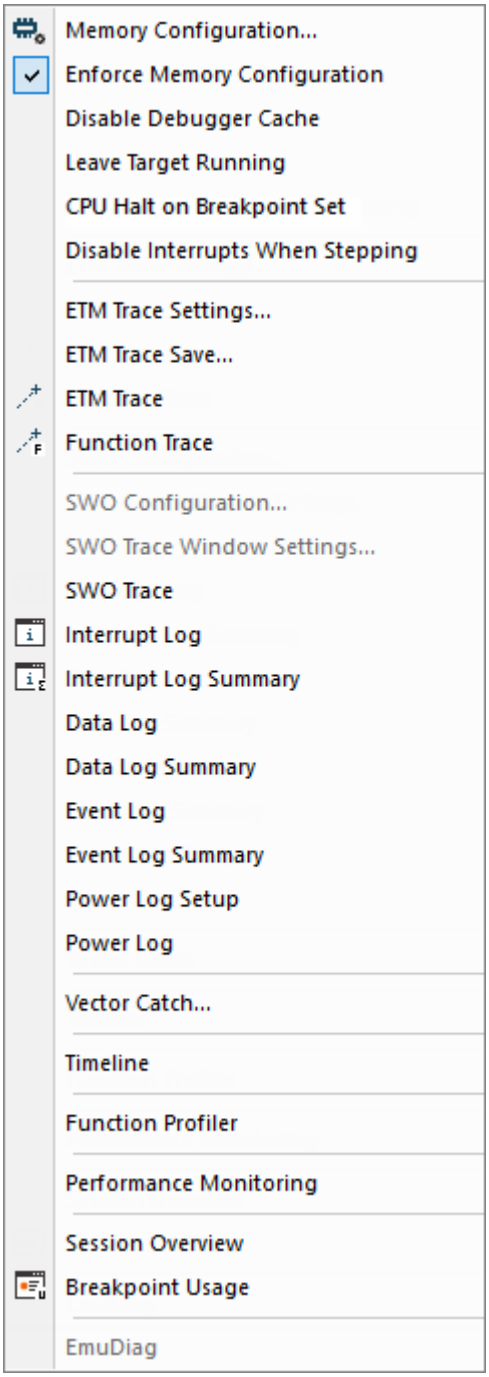
When you are using the C-SPY G+LINK driver, the **G+LINK** menu is added to the menu bar:



For information about G+LINK, see the file `G+ IAR EWARM ICE Driver Pack.pdf`, available in the `arm\bin\G+LinkPro` directory.

I-jet menu

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



Menu commands

These commands are available on the menu:

Memory Configuration
Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.

Enforce Memory Configuration
Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.

Disable Debugger Cache
Disables memory caching and memory range checking in C-SPY.

Normally, C-SPY uses the memory range information in the **Memory Configuration** dialog box both to restrict access to certain parts of target memory and to cache target memory contents for improved C-SPY performance. Under certain rare circumstances, this is not appropriate, and you can choose **Disable Debugger Cache** to turn off the caching and memory range checking completely. All accesses from C-SPY will then result in corresponding accesses to the target system. Some of those circumstances are:

- When memory is remapped at runtime and cannot be specified as a fixed set of ranges.
- When the memory range setup is incorrect or incomplete.

Leave Target Running
Leaves the application running on the target hardware after the debug session is closed.

Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.

CPU Halt on Breakpoint Set
Makes it possible to set a breakpoint in an executing application on hardware that does not support setting breakpoints while running. Setting a breakpoint halts the core, sets the breakpoint, and starts the core again.

Disable Interrupts When Stepping
Ensures that only the stepped statements 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 (I-jet), page 161*.

ETM Trace Save
Displays a dialog box, see *Trace Save dialog box, page 193*.

ETM Trace
Opens the **ETM Trace** window, see *Trace window, page 170*.

Function Trace
Opens the **Function Trace** window, see *Function Trace window, page 178*.

SWO Configuration
Displays a dialog box, see *SWO Configuration dialog box, page 167*.

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 165*.

SWO Trace
Opens the **SWO Trace** window to display the collected trace data, see *Trace window, page 170*.

This menu command is only available when the SWD/SWO interface is used.

Interrupt Log
Opens a window, see *Interrupt Log window, page 318*.

This menu command is only available when the SWD/SWO interface is used.

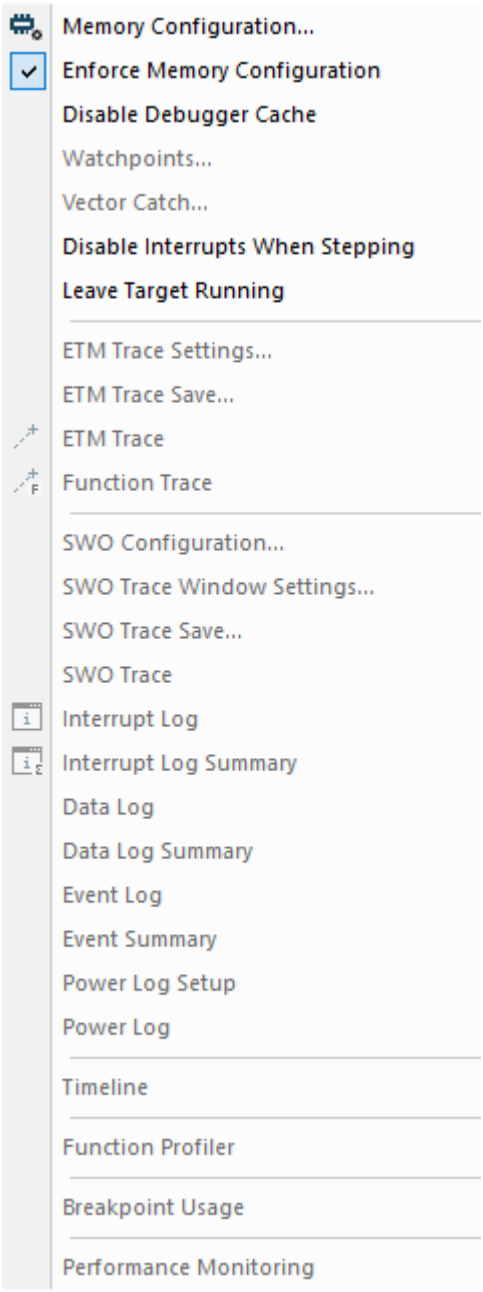
Interrupt Log Summary
Opens a window, see *Interrupt Log Summary window, page 321*.

This menu command is only available when the SWD/SWO interface is used.

Data Log	Opens a window, see *Data Log window, page 201*. 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 203*. This menu command is only available when the SWD/SWO interface is used.
Event Log	Opens a window, see *Event Log window, page 205*.
Event Log Summary	Opens a window, see *Event Log Summary window, page 207*.
Power Log Setup	Opens a window, see *Power Log Setup window, page 249*.
Power Log	Opens a window, see *Power Log window, page 251*.
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 115*. Note that this command is not available for all Arm cores.
Timeline	Opens a window, see *Reference information on application timeline, page 200*. 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 229*.
Performance Monitoring	Opens a window which shows event counters or CPU clock cycles through the Performance Monitoring Unit (PMU), see *Performance Monitoring window, page 240*.
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 101*.
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:

Memory Configuration
Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.

Enforce Memory Configuration
Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.

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.

Watchpoints
Displays a dialog box for setting watchpoints, see *Code breakpoints dialog box, page 102*.

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 115*. Note that this command is not available for all Arm cores.

Disable Interrupts When Stepping
Ensures that only the stepped statements 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.

Leave Target Running
Leaves the application running on the target hardware after the debug session is closed.

Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.

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 163*.

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 193*.

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 170*.

This menu command is only available when using either ETM or J-Link with ETB.

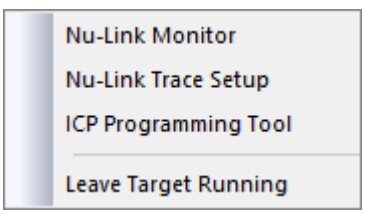
Function Trace
Opens a window which displays the trace data for function calls and function returns, see *Function Trace window, page 178*.

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 167*. 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 165*. 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 193*. 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 170*. This menu command is only available when the SWD/SWO interface is used.
Interrupt Log	Opens a window, see *Interrupt Log window, page 318*. 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 321*. This menu command is only available when the SWD/SWO interface is used.
Data Log	Opens a window, see *Data Log window, page 201*. 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 203*. This menu command is only available when the SWD/SWO interface is used.
Event Log	Opens a window, see *Event Log window, page 205*.
Event Log Summary	Opens a window, see *Event Log Summary window, page 207*.
Power Log Setup	Opens a window, see *Power Log Setup window, page 249*.
Power Log	Opens a window, see *Power Log window, page 251*.
Timeline	Opens a window, see *The application timeline, page 195*. 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 229*.
Breakpoint Usage	Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.
Performance Monitoring	Opens a window which shows event counters or CPU clock cycles through the Performance Monitoring Unit (PMU), see *Performance Monitoring window, page 240*.

Nu-Link menu

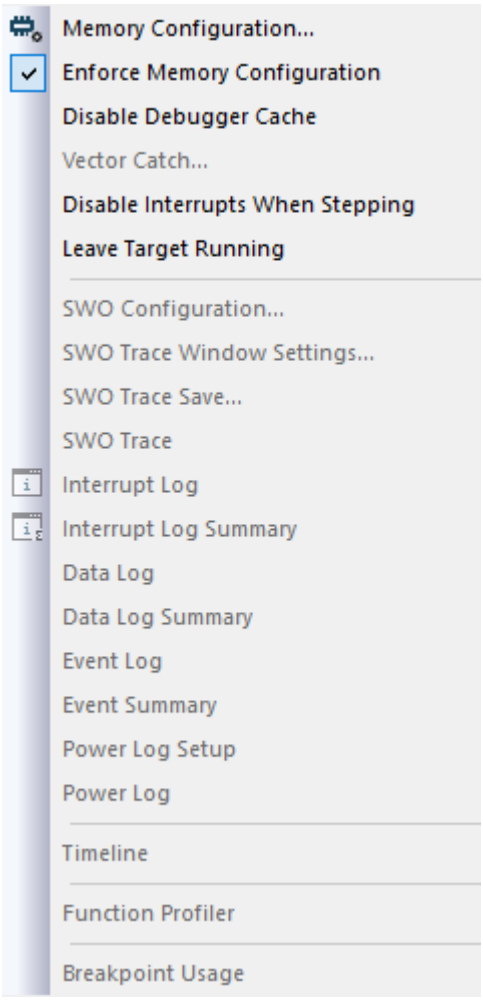
When you are using the C-SPY Nu-Link driver, the **Nu-Link** menu is added to the menu bar:



For information about the Nu-Link driver, contact Nuvoton Technology Corporation.

ST-Link menu

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



Menu commands

These commands are available on the menu:

Memory Configuration	Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.
Enforce Memory Configuration	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
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.
Vector Catch	Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see *Vector Catch dialog box, page 115*. Note that this command is not available for all Arm cores.
Disable Interrupts When Stepping	Ensures that only the stepped statements 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.
Leave Target Running	Leaves the application running on the target hardware after the debug session is closed. Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.
SWO Configuration[1]	Displays a dialog box, see *SWO Configuration dialog box, page 167*.
SWO Trace Window Settings[1]	Displays a dialog box, see *SWO Trace Window Settings dialog box, page 165*.
SWO Trace Save[1]	Displays a dialog box to save the collected trace data to a file, see *Trace Save dialog box, page 193*.
SWO Trace[1]	Opens the **SWO Trace** window to display the collected trace data, see *Trace window, page 170*.
Interrupt Log[1]	Opens a window, see *Interrupt Log window, page 318*.
Interrupt Log Summary[1]	Opens a window, see *Interrupt Log Summary window, page 321*.
Data Log[1]	Opens a window, see *Data Log window, page 201*.
Data Log Summary[1]	Opens a window, see *Data Log Summary window, page 203*.
Event Log	Opens a window, see *Event Log window, page 205*.
Event Log Summary	Opens a window, see *Event Log Summary window, page 207*.
Power Log Setup[1]	Opens a window, see *Power Log Setup window, page 249*.

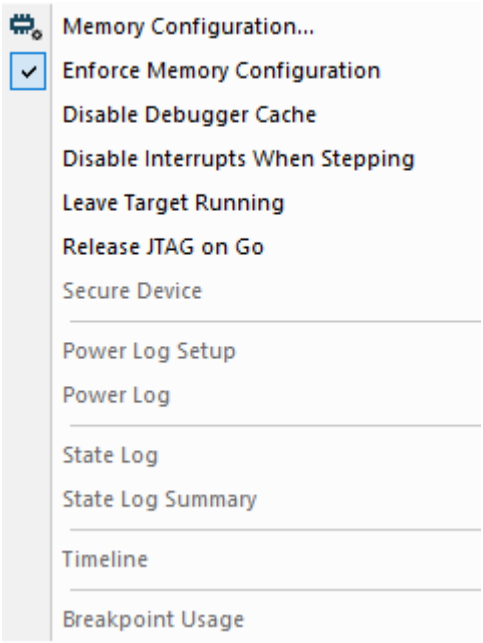
Power Log[1]	Opens a window, see *Power Log window, page 251*.
Timeline[2]	Opens a window, see *The application timeline, page 195*.
Function Profiler	Opens a window which shows timing information for the functions, see *Function Profiler window, page 229*.
Breakpoint Usage	Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

1 Only available when the SWD/SWO interface is used.

2 Available when using either ETM or SWD/SWO.

TI MSP-FET menu

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



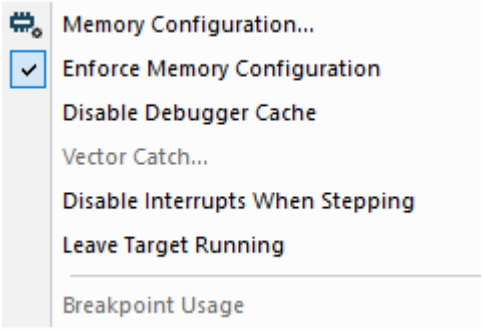
Menu commands

These commands are available on the menu:

Memory Configuration	Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.
Enforce Memory Configuration	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
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 statements 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.
Leave Target Running	Leaves the application running on the target hardware after the debug session is closed.
Release JTAG on Go	 Sets the JTAG drivers in tri-state so that the device is released from JTAG control when the **Go** button is activated.
Secure Device	Activates the flash memory protection lock. After this command has been executed, you must terminate the debug session and then reconnect the power to make it take effect.
Power Log Setup	Opens a window, see *Power Log Setup window, page 249*.
Power Log	Opens a window, see *Power Log window, page 251*.
State Log	Opens a window, see *State Log window, page 255*.
State Log Summary	Opens a window, see *State Log Summary window, page 256*.
Timeline	Opens a window, see *The application timeline, page 195*.
Breakpoint Usage	Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

TI Stellaris menu

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



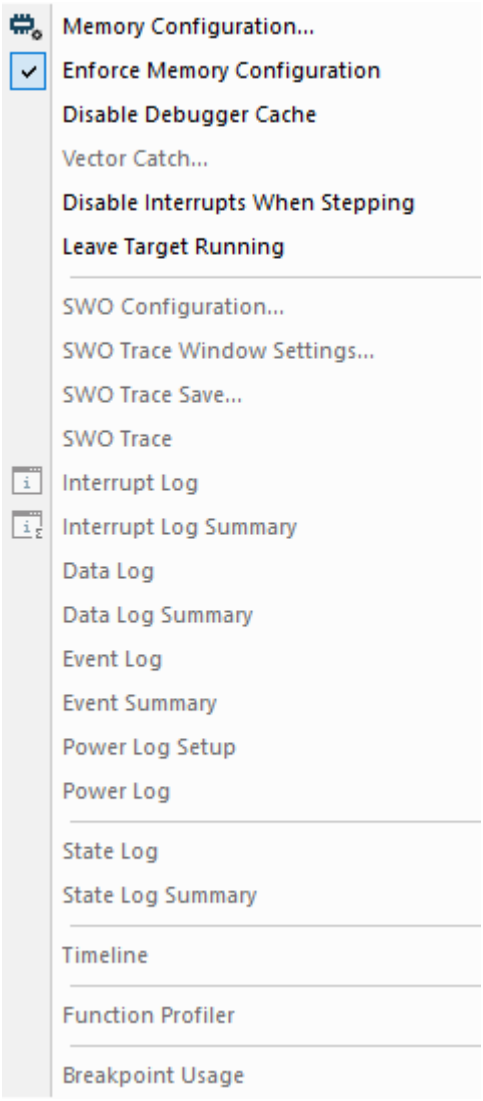
Menu commands

These commands are available on the menu:

Memory Configuration	Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.
Enforce Memory Configuration	Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.
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.
Vector Catch	Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see *Vector Catch dialog box, page 115*. Note that this command is not available for all Arm cores.
Disable Interrupts When Stepping	Ensures that only the stepped statements 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.
Leave Target Running	Leaves the application running on the target hardware after the debug session is closed. Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.
Breakpoint Usage	Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

TI XDS menu

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



Menu commands

These commands are available on the menu:

Memory Configuration
Displays a dialog box, see *Memory Configuration dialog box for C-SPY hardware debugger drivers, page 149*.

Enforce Memory Configuration
Prevents the driver from accessing any memory ranges in the zone `Memory` that have not been defined in the **Memory Configuration** dialog box.

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.

Vector Catch
Opens a dialog box for setting a breakpoint directly on a vector in the interrupt vector table, see *Vector Catch dialog box, page 115*. Note that this command is not available for all Arm cores.

Disable Interrupts When Stepping
Ensures that only the stepped statements 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.

Leave Target Running
Leaves the application running on the target hardware after the debug session is closed.

Because existing breakpoints might not be automatically removed, consider disabling all breakpoints before using this menu command.

SWO Configuration
Displays a dialog box, see *SWO Configuration dialog box, page 167*.

SWO Trace Window Settings
Displays a dialog box, see *SWO Trace Window Settings dialog box, page 165*.

SWO Trace Save
Displays a dialog box to save the collected trace data to a file, see *Trace Save dialog box, page 193*.

SWO Trace
Opens the **SWO Trace** window to display the collected trace data, see *Trace window, page 170*.

Interrupt Log
Opens a window, see *Interrupt Log window, page 318*.

Interrupt Log Summary
Opens a window, see *Interrupt Log Summary window, page 321*.

Data Log
Opens a window, see *Data Log window, page 201*.

Data Log Summary
Opens a window, see *Data Log Summary window, page 203*.

Event Log
Opens a window, see *Event Log window, page 205*.

Event Log Summary
Opens a window, see *Event Log Summary window, page 207*.

Power Log Setup
Opens a window, see *Power Log Setup window, page 249*.

Power Log	Opens a window, see *Power Log window, page 251*.
State Log	Opens a window, see *State Log window, page 255*.
State Log Summary	Opens a window, see *State Log Summary window, page 256*.
Timeline	Opens a window, see *The application timeline, page 195*.
Function Profiler	Opens a window which shows timing information for the functions, see *Function Profiler window, page 229*.
Breakpoint Usage	Opens a window which lists all active breakpoints, see *Breakpoint Usage window, page 101*.

RESOLVING PROBLEMS

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.