



**IAR Embedded
Workbench**

C-SPY® Debugging Guide

for Atmel® Corporation's
AVR Microcontroller Family

COPYRIGHT NOTICE

© 2011–2017 IAR Systems AB.

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

DISCLAIMER

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

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

TRADEMARKS

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Atmel and AVR are registered trademarks of Atmel® Corporation.

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

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

EDITION NOTICE

Sixth edition: April 2017

Part number: UCSAVR-6

This guide applies to version 7.x of IAR Embedded Workbench® for Atmel® Corporation's AVR microcontroller family.

Internal reference: M23, Mym8.0, IMAE.

Brief contents

Tables	19
Preface	21
Part 1. Basic debugging	27
The IAR C-SPY Debugger	29
Getting started using C-SPY	49
Executing your application	63
Variables and expressions	85
Breakpoints	105
Memory and registers	133
Part 2. Analyzing your application	159
Trace	161
The application timeline	175
Profiling	197
Code coverage	207
Power debugging	211
Part 3. Advanced debugging	241
Interrupts	243
C-SPY macros	251
The C-SPY command line utility— <code>cspybat</code>	309

Part 4. Additional reference information	335
Debugger options	337
Additional information on C-SPY drivers	367
Index	381

Contents

Tables	19
Preface	21
Who should read this guide	21
Required knowledge	21
What this guide contains	21
Part 1. Basic debugging	21
Part 2. Analyzing your application	22
Part 3. Advanced debugging	22
Part 4. Additional reference information	22
Other documentation	23
User and reference guides	23
The online help system	24
Web sites	24
Document conventions	24
Typographic conventions	25
Naming conventions	25
Part I. Basic debugging	27
The IAR C-SPY Debugger	29
Introduction to C-SPY	29
An integrated environment	29
General C-SPY debugger features	30
RTOS awareness	31
Debugger concepts	32
C-SPY and target systems	32
The debugger	33
The target system	33
The application	33
C-SPY debugger systems	33
The ROM-monitor program	34

Third-party debuggers	34
C-SPY plugin modules	34
C-SPY drivers overview	35
Differences between the C-SPY drivers	35
The IAR C-SPY Simulator	36
The C-SPY Atmel-ICE driver	37
Features	37
Communication overview	38
Hardware installation	38
The C-SPY Power Debugger driver	39
Features	39
Communication overview	39
Hardware installation	40
The C-SPY JTAGICE3 driver	41
Features	41
Communication overview	42
Hardware installation	42
The C-SPY JTAGICE mkII/Dragon driver	43
Features	43
Communication overview	44
Hardware installation	44
The C-SPY AVR ONE! driver	45
Features	45
Communication overview	46
Hardware installation	46
Getting started using C-SPY	49
Setting up C-SPY	49
Setting up for debugging	49
Executing from reset	50
Using a setup macro file	50
Selecting a device description file	51
Loading plugin modules	51

Starting C-SPY	51
Starting a debug session	52
Loading executable files built outside of the IDE	52
Starting a debug session with source files missing	52
Loading multiple images	53
Editing in C-SPY windows	54
Adapting for target hardware	55
Modifying a device description file	55
Initializing target hardware before C-SPY starts	55
Reference information on starting C-SPY	56
C-SPY Debugger main window	56
Images window	60
Get Alternative File dialog box	62
Executing your application	63
Introduction to application execution	63
Briefly about application execution	63
Source and disassembly mode debugging	63
Single stepping	64
Troubleshooting slow stepping speed	66
Running the application	67
Highlighting	68
Viewing the call stack	68
Terminal input and output	69
Debug logging	69
Reference information on application execution	70
Disassembly window	71
Call Stack window	75
Terminal I/O window	77
Terminal I/O Log File dialog box	78
Debug Log window	79
Log File dialog box	80
Report Assert dialog box	81
Autostep settings dialog box	82

Cores window	82
Variables and expressions	85
Introduction to working with variables and expressions	85
Briefly about working with variables and expressions	85
C-SPY expressions	86
Limitations on variable information	88
Working with variables and expressions	89
Using the windows related to variables and expressions	89
Viewing assembler variables	89
Reference information on working with variables and expressions	90
Auto window	91
Locals window	93
Watch window	95
Statics window	97
Quick Watch window	100
Symbols window	102
Resolve Symbol Ambiguity dialog box	104
Breakpoints	105
Introduction to setting and using breakpoints	105
Reasons for using breakpoints	105
Briefly about setting breakpoints	105
Breakpoint types	106
Breakpoint icons	108
Breakpoints in the C-SPY simulator	108
Breakpoints in the C-SPY hardware debugger drivers	108
Breakpoint consumers	110
Setting breakpoints	111
Various ways to set a breakpoint	112
Toggling a simple code breakpoint	112
Setting breakpoints using the dialog box	112
Setting a data breakpoint in the Memory window	113
Setting breakpoints using system macros	114

Useful breakpoint hints	115
Reference information on breakpoints	116
Breakpoints window	117
Breakpoint Usage window	119
Code breakpoints dialog box	120
Log breakpoints dialog box	121
Data breakpoints dialog box	123
Data Log breakpoints dialog box	125
Immediate breakpoints dialog box	126
Complex breakpoints dialog box	127
Enter Location dialog box	130
Resolve Source Ambiguity dialog box	131
Memory and registers	133
Introduction to monitoring memory and registers	133
Briefly about monitoring memory and registers	133
C-SPY memory zones	134
Memory configuration for the C-SPY simulator	135
Monitoring memory and registers	135
Defining application-specific register groups	136
Monitoring stack usage	136
Reference information on memory and registers	139
Memory window	140
Memory Save dialog box	143
Memory Restore dialog box	144
Fill dialog box	145
Symbolic Memory window	146
Stack window	149
Registers window	153
Register User Groups Setup window	156

Part 2. Analyzing your application	159
Trace	161
Introduction to using trace	161
Reasons for using trace	161
Briefly about trace	161
Requirements for using trace	162
Collecting and using trace data	162
Getting started with trace	162
Trace data collection using breakpoints	162
Searching in trace data	163
Browsing through trace data	163
Reference information on trace	164
Trace window	164
Function Trace window	167
Trace Start breakpoints dialog box	168
Trace Stop breakpoints dialog box	169
Trace Expressions window	170
Find in Trace dialog box	172
Find in Trace window	173
The application timeline	175
Introduction to analyzing your application’s timeline	175
Briefly about analyzing the timeline	175
Requirements for timeline support	177
Analyzing your application’s timeline	177
Displaying a graph in the Timeline window	177
Navigating in the graphs	178
Analyzing performance using the graph data	178
Getting started using data logging	179
Reference information on application timeline	180
Timeline window—Call Stack graph	181
Timeline window—Data Log graph	184
Data Log window	188

Data Log Summary window	191
Viewing Range dialog box	194
Profiling	197
Introduction to the profiler	197
Reasons for using the profiler	197
Briefly about the profiler	197
Requirements for using the profiler	198
Using the profiler	198
Getting started using the profiler on function level	199
Analyzing the profiling data	199
Getting started using the profiler on instruction level	201
Reference information on the profiler	202
Function Profiler window	202
Code coverage	207
Introduction to code coverage	207
Reasons for using code coverage	207
Briefly about code coverage	207
Requirements and restrictions for using code coverage	207
Reference information on code coverage	207
Code Coverage window	208
Power debugging	211
Introduction to power debugging	211
Reasons for using power debugging	211
Briefly about power debugging	211
Requirements and restrictions for power debugging	213
Optimizing your source code for power consumption	213
Waiting for device status	213
Software delays	213
Low-power mode diagnostics	214
CPU frequency	214
Detecting mistakenly unattended peripherals	215
Peripheral units in an event-driven system	215

Finding conflicting hardware setups	216
Analog interference	216
Debugging in the power domain	217
Displaying a power profile and analyzing the result	217
Detecting unexpected power usage during application execution ...	219
Changing the graph resolution	219
Reference information on power debugging	219
Power Log Setup window	220
Power Debugging Settings	222
Timeline window—Power graph	223
Power Log window	226
State Log Setup window	229
State Log window	231
State Log Summary window	233
Timeline window—State Log graph	235
Part 3. Advanced debugging	241
Interrupts	243
Introduction to interrupts	243
Briefly about the interrupt simulation system	243
Interrupt characteristics	244
C-SPY system macros for interrupt simulation	245
Target-adapting the interrupt simulation system	245
Using the interrupt system	246
Simulating a simple interrupt	246
Reference information on interrupts	247
Interrupts dialog box	248
C-SPY macros	251
Introduction to C-SPY macros	251
Reasons for using C-SPY macros	251
Briefly about using C-SPY macros	252
Briefly about setup macro functions and files	252

Briefly about the macro language	252
Using C-SPY macros	253
Registering C-SPY macros—an overview	254
Executing C-SPY macros—an overview	254
Registering and executing using setup macros and setup files	255
Executing macros using Quick Watch	255
Executing a macro by connecting it to a breakpoint	256
Aborting a C-SPY macro	257
Reference information on the macro language	258
Macro functions	258
Macro variables	258
Macro parameters	259
Macro strings	259
Macro statements	260
Formatted output	261
Reference information on reserved setup macro function names	263
execUserPreload	263
execUserExecutionStarted	264
execUserExecutionStopped	264
execUserSetup	264
execUserPreReset	265
execUserReset	265
execUserExit	265
Reference information on C-SPY system macros	265
__abortLaunch	267
__cancelAllInterrupts	268
__cancelInterrupt	268
__clearBreak	269
__closeFile	269
__delay	269
__disableInterrupts	270
__driverType	270
__enableInterrupts	271

__evaluate	271
__fillMemory8	272
__fillMemory16	272
__fillMemory32	273
__getCycleCounter	274
__isBatchMode	275
__loadImage	275
__memoryRestore	276
__memoryRestoreFromFile	277
__memorySave	277
__memorySaveToFile	278
__messageBoxYesCancel	279
__messageBoxYesNo	279
__openFile	280
__orderInterrupt	281
__readFile	282
__readFileByte	283
__readMemory8, __readMemoryByte	283
__readMemory16	284
__readMemory32	284
__registerMacroFile	285
__resetFile	285
__setCodeBreak	286
__setComplexBreak	287
__setDataBreak	289
__setLogBreak	291
__setSimBreak	292
__setTraceStartBreak	293
__setTraceStopBreak	294
__sourcePosition	295
__strFind	295
__subString	296
__targetDebuggerVersion	296
__toLower	297

__toString	297
__toUpper	298
__unloadImage	298
__writeFile	299
__writeFileByte	299
__writeMemory8, __writeMemoryByte	300
__writeMemory16	300
__writeMemory32	301
Graphical environment for macros	301
Macro Registration window	302
Debugger Macros window	304
Macro Quicklaunch window	306
The C-SPY command line utility—cspybat	309
Using C-SPY in batch mode	309
Starting cspybat	309
Output	310
Invocation syntax	310
Summary of C-SPY command line options	311
General cspybat options	311
Options available for all C-SPY drivers	312
Options available for the simulator driver	313
Options available for all C-SPY hardware debugger drivers	313
Options available for the C-SPY Power Debugger driver	313
Options available for the C-SPY JTAGICE mkII driver, the C-SPY Dragon driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver	313
Options available for the C-SPY JTAGICE mkII driver, the C-SPY Dragon driver, the C-SPY Atmel-ICE driver, the C-SPY Power Debugger driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver	314
Options available for the C-SPY JTAGICE mkII driver and the C-SPY Dragon driver	314
Options available for the C-SPY Atmel-ICE driver, the	

C-SPY Power Debugger driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver	314
Options available for the C-SPY JTAGICE mkII driver and the C-SPY Dragon driver	314
Options available for the C-SPY Dragon driver	315
Reference information on C-SPY command line options ...	315
--64bit_doubles	315
--64k_flash	315
--attach_to_running_target	315
--avrone_jtag_clock	316
--backend	316
--code_coverage_file	317
--cpu	317
--cycles	318
--debugfile	318
--disable_internal_eeprom	319
--disable_interrupts	319
--download_only	319
--drv_communication	319
--drv_communication_log	320
--drv_debug_port	320
--drv_download_data	321
--drv_dragon	321
--drv_power_debugger	322
--drv_preserve_app_section	322
--drv_preserve_boot_section	322
--drv_set_exit_breakpoint	323
--drv_set_getchar_breakpoint	323
--drv_set_putchar_breakpoint	324
--drv_suppress_download	324
--drv_use_PDI	325
--drv_verify_download	325
--eeprom_size	325
--enhanced_core	326

-f	326
--function_profiling	326
--jtagice_clock	327
--jtagice_do_hardware_reset	327
--jtagice_leave_timers_running	328
--jtagice_preserve_eeprom	328
--jtagice_restore_fuse	328
--jtagicemkII_use_software_breakpoints	329
--leave_target_running	329
--macro	330
--macro_param	330
-p	331
--plugin	331
--program_fuses_after_download	332
--silent	333
--timeout	333
-v	333

Part 4. Additional reference information 335

Debugger options 337

Setting debugger options 337

Reference information on general debugger options 338

Setup 339

Images 340

Plugins 341

Reference information on C-SPY hardware debugger driver options 342

Atmel-ICE 1 342

Atmel-ICE 2 345

Communication 346

Extra Options 347

AVR ONE! 1 348

AVR ONE! 2 350

JTAGICE3 1	351
JTAGICE3 2	353
JTAGICE mkII 1	354
JTAGICE mkII 2	357
Serial Port	358
Dragon 1	359
Dragon 2	361
Power Debugger 1	362
Power Debugger 2	364
Third-Party Driver options	365
Additional information on C-SPY drivers	367
Reference information on C-SPY driver menus	367
<i>C-SPY driver</i>	367
Simulator menu	368
JTAGICE mkII menu	369
Dragon menu	369
Atmel-ICE menu	370
JTAGICE3 menu	371
AVR ONE! menu	372
Power Debugger menu	373
Reference information on the C-SPY hardware debugger	
drivers	374
Fuse Handler dialog box	375
Fuse Handler dialog box	377
Resolving problems	379
No contact with the target hardware	379
Index	381

Tables

1: Typographic conventions used in this guide	25
2: Naming conventions used in this guide	25
3: Driver differences	35
4: C-SPY assembler symbols expressions	87
5: Handling name conflicts between hardware registers and assembler labels	87
6: Available breakpoints in C-SPY hardware debugger drivers	109
7: C-SPY macros for breakpoints	114
8: Support for timeline information	177
9: Project options for enabling the profiler	199
10: Project options for enabling code coverage	208
11: Timer interrupt settings	247
12: Examples of C-SPY macro variables	259
13: Summary of system macros	265
14: __cancelInterrupt return values	268
15: __disableInterrupts return values	270
16: __driverType return values	270
17: __enableInterrupts return values	271
18: __evaluate return values	271
19: __isBatchMode return values	275
20: __loadImage return values	275
21: __messageBoxYesCancel return values	279
22: __messageBoxYesNo return values	279
23: __openFile return values	280
24: __readFile return values	282
25: __setCodeBreak return values	286
26: __set Complex Break return values	289
27: __setDataBreak return values	290
28: __setLogBreak return values	291
29: __setSimBreak return values	292
30: __setTraceStartBreak return values	293
31: __setTraceStopBreak return values	294

32: __sourcePosition return values	295
33: __unloadImage return values	299
34: cspybat parameters	310
35: Options specific to the C-SPY drivers you are using	337

Preface

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

Who should read this guide

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

REQUIRED KNOWLEDGE

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

- The architecture and instruction set of the Atmel AVR microcontroller (refer to the chip manufacturer's documentation)
- The C or C++ programming language
- Application development for embedded systems
- The operating system of your host computer.

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

What this guide contains

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

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

PART I. BASIC DEBUGGING

- *The IAR C-SPY Debugger* introduces you to the C-SPY debugger and to the concepts that are related to debugging in general and to C-SPY in particular. The chapter also introduces the various C-SPY drivers. The chapter briefly shows the difference in functionality that the various C-SPY drivers provide.
- *Getting started using C-SPY* helps you get started using C-SPY, which includes setting up, starting, and adapting C-SPY for target hardware.

- *Executing your application* describes the conceptual differences between source and disassembly mode debugging, the facilities for executing your application, and finally, how you can handle terminal input and output.
- *Variables and expressions* describes the syntax of the expressions and variables used in C-SPY, as well as the limitations on variable information. The chapter also demonstrates the various methods for monitoring variables and expressions.
- *Breakpoints* describes the breakpoint system and the various ways to set breakpoints.
- *Memory and registers* shows how you can examine memory and registers.

PART 2. ANALYZING YOUR APPLICATION

- *Trace* describes how you can inspect the program flow up to a specific state using trace data.
- *The application timeline* describes the **Timeline** window, and how to use the information in it to analyze your application's behavior.
- *Profiling* describes how the profiler can help you find the functions in your application source code where the most time is spent during execution.
- *Code coverage* describes how the code coverage functionality can help you verify whether all parts of your code have been executed, thus identifying parts which have not been executed.
- *Power debugging* describes techniques for power debugging and how you can use C-SPY to find source code constructions that result in unexpected power consumption.

PART 3. ADVANCED DEBUGGING

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

PART 4. ADDITIONAL REFERENCE INFORMATION

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

Other documentation

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

USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, is available in the booklet Quick Reference (available in the product box) and the *Installation and Licensing Guide*.
- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide for AVR*.
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide for AVR*.
- Programming for the IAR C/C++ Compiler for AVR, is available in the *IAR C/C++ Compiler Reference Guide for AVR*.
- Using the IAR XLINK Linker, the IAR XAR Library Builder, and the IAR XLIB Librarian, is available in the *IAR Linker and Library Tools Reference Guide*.
- Programming for the IAR Assembler for AVR, is available in the *AVR® IAR Assembler Reference Guide*.
- Performing a static analysis using C-STAT and the required checks, is available in the *C-STAT Static Analysis Guide*.
- Developing safety-critical applications using the MISRA C guidelines, is available in the *IAR Embedded Workbench® MISRA C:2004 Reference Guide* or the *IAR Embedded Workbench® MISRA C:1998 Reference Guide*.
- Porting application code and projects created with a previous version of the IAR Embedded Workbench for AVR, is available in the *IAR Embedded Workbench® Migration Guide*.

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

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about project management, editing, and building in the IDE
- Information about debugging using the IAR C-SPY® Debugger
- Reference information about the menus, windows, and dialog boxes in the IDE
- Compiler reference information
- Keyword reference information for the DLIB library functions. To obtain reference information for a function, select the function name in the editor window and press F1. Note that if you select a function name in the editor window and press F1 while using the CLIB C standard library, you will get reference information for the DLIB C/EC++ standard library.

WEB SITES

Recommended web sites:

- The Atmel® Corporation web site, www.atmel.com, that contains information and news about the Atmel AVR microcontrollers.
- The IAR Systems web site, www.iar.com, that holds application notes and other product information.
- The web site of the C standardization working group, www.open-std.org/jtc1/sc22/wg14.
- The web site of the C++ Standards Committee, www.open-std.org/jtc1/sc22/wg21.
- 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 Systems documentation, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

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

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:





Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> • Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file.
[option]	An optional part of a directive, where [and] are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
{option}	A mandatory part of a directive, where { and } are not part of the actual directive, but any [,], {, or } are part of the directive syntax.
[option]	An optional part of a command.
[a b c]	An optional part of a command with alternatives.
{a b c}	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
<i>italic</i>	<ul style="list-style-type: none"> • A cross-reference within this guide or to another guide. • Emphasis.
...	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the IAR Embedded Workbench® IDE interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.
	Identifies warnings.

Table 1: Typographic conventions used in this guide

NAMING CONVENTIONS

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

Brand name	Generic term
IAR Embedded Workbench® for AVR	IAR Embedded Workbench®

Table 2: Naming conventions used in this guide

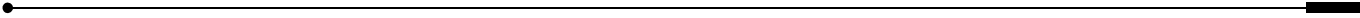
Brand name	Generic term
IAR Embedded Workbench® IDE for AVR	the IDE
IAR C-SPY® Debugger for AVR	C-SPY, the debugger
IAR C-SPY® Simulator	the simulator
IAR C/C++ Compiler™ for AVR	the compiler
IAR Assembler™ for AVR	the assembler
IAR XLINK Linker™	XLINK, the linker
IAR XAR Library Builder™	the library builder
IAR XLIB Librarian™	the librarian
IAR DLIB Runtime Environment™	the DLIB runtime environment
IAR CLIB Runtime Environment™	the CLIB runtime environment

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

Part I. Basic debugging

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

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





The IAR C-SPY Debugger

- Introduction to C-SPY
- Debugger concepts
- C-SPY drivers overview
- The IAR C-SPY Simulator
- The C-SPY Atmel-ICE driver
- The C-SPY Power Debugger driver
- The C-SPY JTAGICE3 driver
- The C-SPY JTAGICE mkII/Dragon driver
- The C-SPY AVR ONE! driver

Introduction to C-SPY

These topics are covered:

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

AN INTEGRATED ENVIRONMENT

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

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

watch properties, window layouts, and register groups will be preserved between your debug sessions.

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

GENERAL C-SPY DEBUGGER FEATURES

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

C-SPY offers these general features:

- Source and disassembly level debugging

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

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

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

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

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

The compiler generates extensive call stack information. This allows the debugger to show, without any runtime penalty, the complete stack of function calls wherever the

program counter is. You can select any function in the call stack, and for each function you get valid information for local variables and available registers.

- Powerful macro system

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

Additional general C-SPY debugger features

This list shows some additional features:

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

RTOS AWARENESS

C-SPY supports RTOS-aware debugging.

These operating systems are currently supported:

- Micrium uC/OS-II
- OSEK Run Time Interface (ORTI)

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

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

A loaded plugin will add its own menu, set of windows, and buttons when a debug session is started (provided that the RTOS is linked with the application). For

information about other RTOS awareness plugin modules, refer to the manufacturer of the plugin module.

Debugger concepts

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

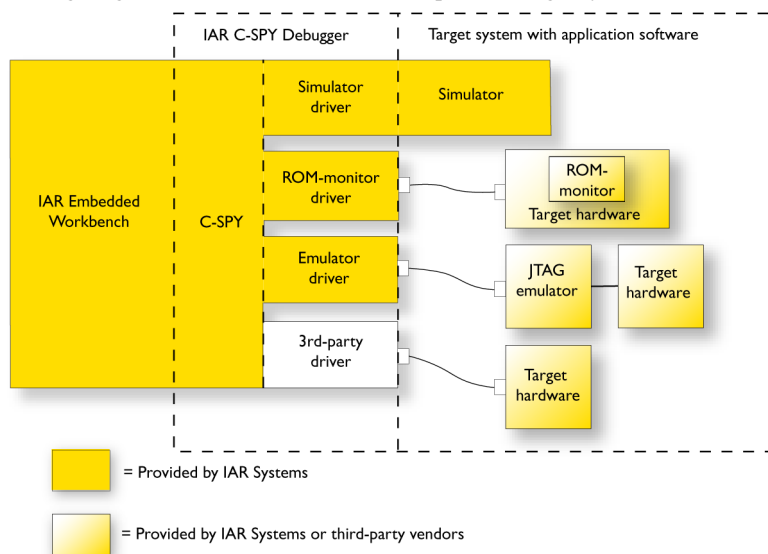
These topics are covered:

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

C-SPY AND TARGET SYSTEMS

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

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



THE DEBUGGER

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

THE TARGET SYSTEM

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

THE APPLICATION

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

C-SPY DEBUGGER SYSTEMS

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

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

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

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

THE ROM-MONITOR PROGRAM

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

THIRD-PARTY DEBUGGERS

You can use a third-party debugger together with the IAR Systems toolchain as long as the third-party debugger can read any of the output formats provided by XLINK, such as UBROF, ELF/DWARF, COFF, Intel-extended, Motorola, or any other available format. For information about which format to use with a third-party debugger, see the user documentation supplied with that tool.

C-SPY PLUGIN MODULES

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

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

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

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

C-SPY drivers overview

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

- Simulator
- AVR® Atmel-ICE
- AVR® Atmel Power Debugger
- AVR® JTAGICE3
- AVR® JTAGICE mkII/AVR® Dragon
- AVR® AVR ONE!

Note: In addition to the drivers supplied with IAR Embedded Workbench, you can also load debugger drivers supplied by a third-party vendor; see *Third-Party Driver options*, page 365.

DIFFERENCES BETWEEN THE C-SPY DRIVERS

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

Feature	Simulator	Atmel-ICE	Power Debugger	JTAGICE3	JTAGICE mkII/ Dragon	AVR ONE!
Code breakpoints ¹	Yes	Yes ¹	Yes ¹	Yes ¹	Yes ¹	Yes ¹
Data breakpoints ¹	Yes	Yes	Yes	Yes	Yes ¹	Yes
Execution in real time ¹	—	Yes	Yes	Yes	Yes	Yes
Zero memory footprint ¹	Yes	Yes	Yes	Yes	Yes	Yes
Simulated interrupts ¹	Yes	—	—	—	—	—
Real interrupts ¹	—	Yes	Yes	Yes	Yes	Yes
Data logging ¹	Yes	—	—	—	—	—
Cycle counter ¹	Yes	—	—	—	—	—
Code coverage ¹	Yes	—	—	—	—	—
Data coverage ¹	Yes	—	—	—	—	—

Table 3: Driver differences

Feature	Simulator	Atmel-ICE	Power Debugger	JTAGICE3	JTAGICE mkII/ Dragon	AVR ONE!
Function/ instruction profiler ¹	Yes	—	—	—	—	—
Trace ¹	Yes	—	—	—	—	—
Power debugging ¹	Yes	—	Yes	—	—	—

Table 3: Driver differences (Continued)

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

The IAR C-SPY Simulator

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

The C-SPY Simulator supports:

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

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

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

The C-SPY Atmel-ICE driver

The C-SPY Atmel-ICE driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY Atmel-ICE driver, C-SPY can connect to Atmel-ICE, Atmel Power Debugger, or EDBG. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

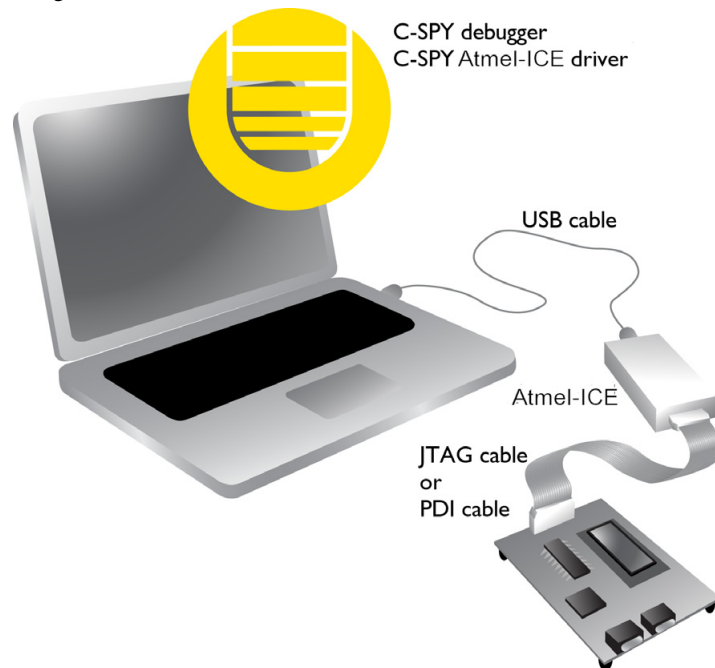
FEATURES

In addition to the general features of C-SPY, the Atmel-ICE driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via USB
- Fuse handler.

COMMUNICATION OVERVIEW

The C-SPY Atmel-ICE driver uses the USB port to communicate with Atmel-ICE. Atmel-ICE communicates with the hardware interface—for example JTAG, PDI, debugWIRE, ISP, TPI, or UPDI—on the microcontroller.



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

HARDWARE INSTALLATION

For information about the hardware installation, see the *AVR® Atmel-ICE User Guide* from Atmel® Corporation. This power-up sequence is recommended to ensure proper communication between the target board, Atmel-ICE, and C-SPY:

- 1 Power up the target board.
- 2 Power up Atmel-ICE.
- 3 Start the C-SPY debug session.

To enable the hardware interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded

Workbench IDE or a similar tool capable of programming the fuses to check and program these bits. For more information, see the *Fuse Handler dialog box*, page 375.

The C-SPY Power Debugger driver

The C-SPY Power Debugger driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY Power Debugger driver, C-SPY can connect to Atmel Power Debugger. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

FEATURES

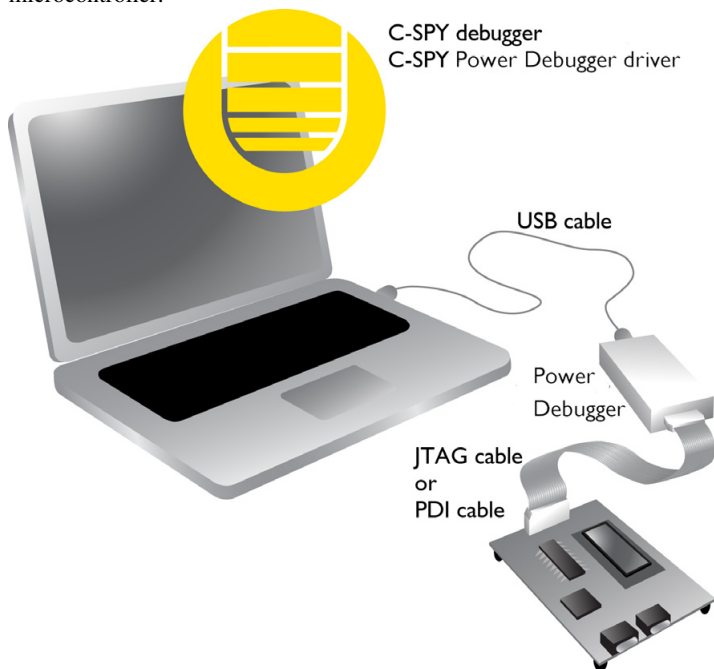
In addition to the general features of C-SPY, the Power Debugger driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via USB
- Fuse handler.

COMMUNICATION OVERVIEW

The C-SPY Power Debugger driver uses the USB port to communicate with the Atmel Power Debugger. The Atmel Power Debugger communicates with the hardware

interface—for example JTAG, PDI, debugWIRE, ISP, TPI, or UPDI—on the microcontroller.



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

HARDWARE INSTALLATION

For information about the hardware installation, see the Atmel® Corporation web site www.atmel.com. This power-up sequence is recommended to ensure proper communication between the target board, the Atmel Power Debugger, and C-SPY:

- 1 Power up the target board.
- 2 Power up the Atmel Power Debugger.
- 3 Start the C-SPY debug session.

To enable the hardware interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded Workbench IDE or a similar tool capable of programming the fuses to check and program these bits. For more information, see the *Fuse Handler dialog box*, page 375.

The C-SPY JTAGICE3 driver

The C-SPY JTAGICE3 driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY JTAGICE3 driver, C-SPY can connect to JTAGICE3. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

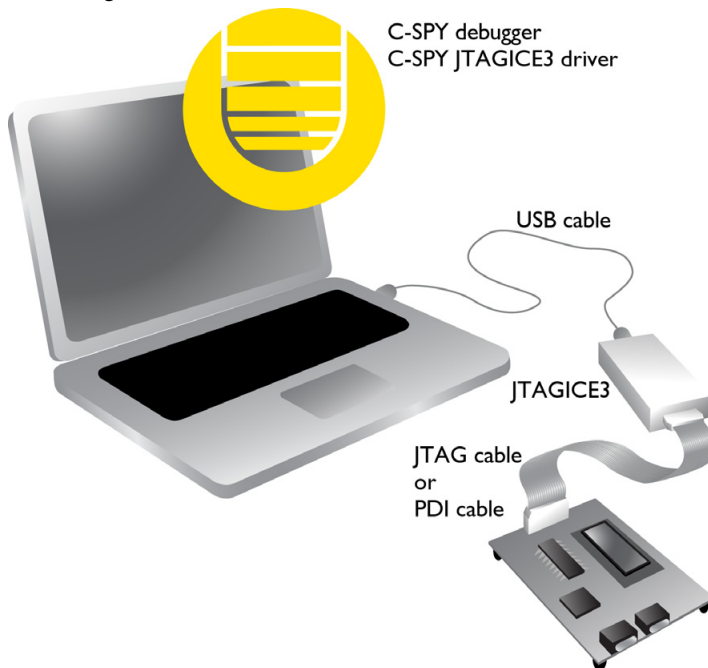
FEATURES

In addition to the general features of C-SPY, the JTAGICE3 driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via USB
- Fuse handler.

COMMUNICATION OVERVIEW

The C-SPY JTAGICE3 driver uses the USB port to communicate with Atmel JTAGICE3. JTAGICE3 communicates with the hardware interface—for example JTAG, PDI, debugWIRE, or ISP—on the microcontroller.



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

HARDWARE INSTALLATION

For information about the hardware installation, see the *AVR® JTAGICE3 User Guide* from Atmel® Corporation. This power-up sequence is recommended to ensure proper communication between the target board, JTAGICE3, and C-SPY:

- 1 Power up the target board.
- 2 Power up JTAGICE3.
- 3 Start the C-SPY debug session.

To enable the hardware interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded

Workbench IDE or a similar tool capable of programming the fuses to check and program these bits. For more information, see the *Fuse Handler dialog box*, page 375.

The C-SPY JTAGICE mkII/Dragon driver

The C-SPY JTAGICE mkII driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY JTAGICE mkII driver, C-SPY can connect to JTAGICE mkII and Dragon. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

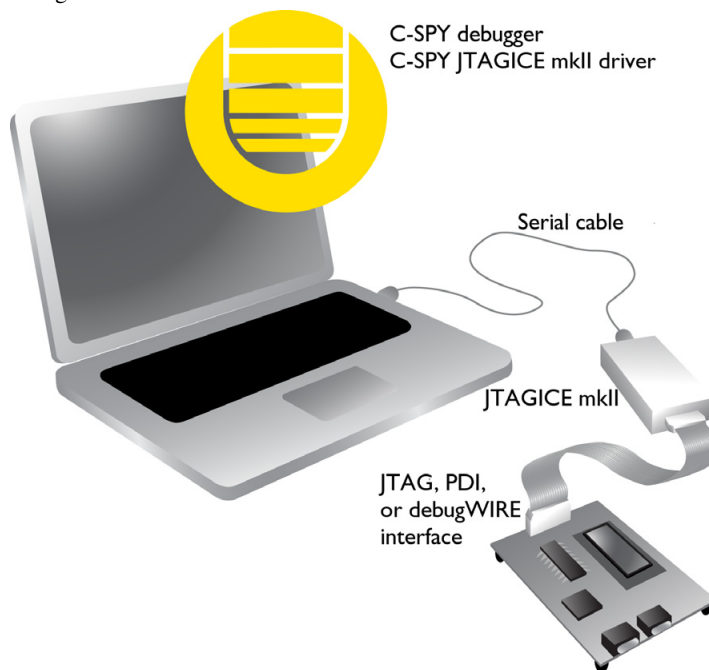
FEATURES

In addition to the general features of C-SPY, the JTAGICE mkII driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints, for devices that support software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via the serial port or USB
- Fuse handler.

COMMUNICATION OVERVIEW

The C-SPY JTAGICE mkII driver uses the serial port to communicate with Atmel AVR JTAGICE mkII. JTAGICE mkII communicates with the JTAG, the PDI, or the debugWIRE interface on the microcontroller.



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

HARDWARE INSTALLATION

For information about the hardware installation, see the *AVR® JTAGICE mkII User Guide* from Atmel® Corporation. The following power-up sequence is recommended to ensure proper communication between the target board, JTAGICE mkII, and C-SPY:

- 1 Power up the target board.
- 2 Power up JTAGICE mkII.
- 3 Start the C-SPY debug session.

To enable the JTAG interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded

Workbench IDE or a similar tool capable of programming the fuses to check and program these bits. For more information, see the *Fuse Handler dialog box*, page 375.

The C-SPY AVR ONE! driver

The C-SPY AVR ONE! driver is automatically installed during the installation of IAR Embedded Workbench. Using the C-SPY AVR ONE! driver, C-SPY can connect to AVR ONE!. Many of the AVR microcontrollers have built-in, on-chip debug support. Because the hardware debugger logic is built into the microcontroller, no ordinary ROM-monitor program or extra specific hardware is needed to make the debugging work.

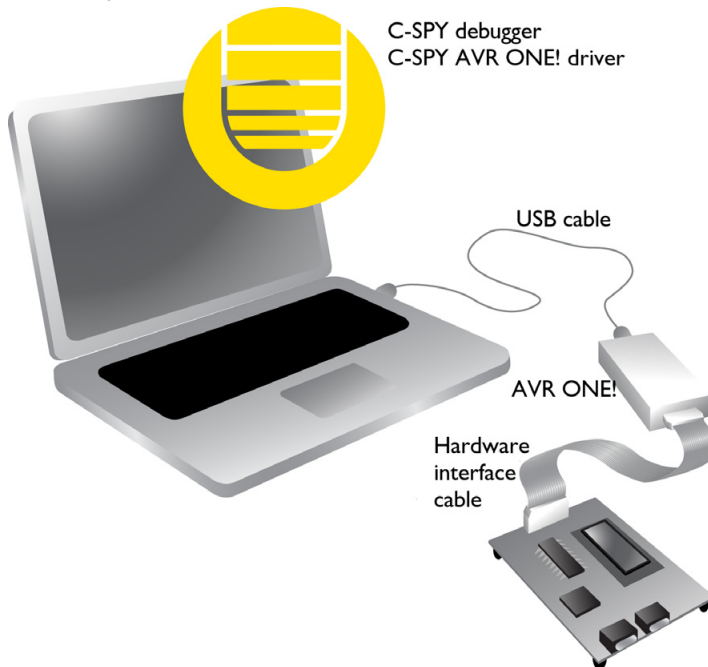
FEATURES

In addition to the general features of C-SPY, the AVR ONE! driver also provides:

- Execution in real time with full access to the microcontroller
- Use of the available hardware breakpoints on the target device and unlimited use of software breakpoints
- Zero memory footprint on the target system
- Built-in flash loader
- Communication via USB
- Fuse handler.

COMMUNICATION OVERVIEW

The C-SPY AVR ONE! driver uses the USB port to communicate with Atmel AVR ONE!. AVR ONE! communicates with the hardware interface—for example JTAG, PDI, debugWIRE, or ISP—on the microcontroller.



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

HARDWARE INSTALLATION

For information about the hardware installation, see the *AVR® AVR ONE! User Guide* from Atmel® Corporation. This power-up sequence is recommended to ensure proper communication between the target board, AVR ONE!, and C-SPY:

- 1 Power up the target board.
- 2 Power up AVR ONE!.
- 3 Start the C-SPY debug session.

To enable the hardware interface on the microcontroller, the JTAG and OCD fuse bits must be enabled. Use the **Fuse Handler** dialog box available in the IAR Embedded

Workbench IDE or a similar tool capable of programming the fuses to check and program these bits. For more information, see the *Fuse Handler dialog box*, page 377.

Getting started using C-SPY

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

Setting up C-SPY

These tasks are covered:

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

SETTING UP FOR DEBUGGING

- 1** Before you start C-SPY, choose **Project>Options>Debugger>Setup** and select the C-SPY driver that matches your debugger system: simulator or a hardware debugger system.
- 2** In the **Category** list, select the appropriate C-SPY driver and make your settings.
For information about these options, see *Debugger options*, page 337.
- 3** Click **OK**.
- 4** Choose **Tools>Options** to open the **IDE Options** dialog box:
 - Select **Debugger** to configure the debugger behavior
 - Select **Stack** to configure the debugger's tracking of stack usage.

For more information about these options, see the *IDE Project Management and Building Guide for AVR*.

See also *Adapting for target hardware*, page 55.

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 then contain the regular hardware reset address at each reset. Or, if you have selected the option **Get reset address from UBROF**, the program counter will contain the `__program_start` label.

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

Note: This message will never be displayed in the C-SPY Simulator, where breakpoints are unlimited.

USING A SETUP MACRO FILE

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

For more information about setup macro files and functions, see *Introduction to C-SPY macros*, page 251. For an example of how to use a setup macro file, see *Initializing target hardware before C-SPY starts*, page 55.

To register a setup macro file:

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

SELECTING A DEVICE DESCRIPTION FILE

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

A default device description file 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 are provided in the `AVR\config` directory and they have the filename extension `ddf`.

For more information about device description files, see *Adapting for target hardware*, page 55.

To override the default device description file:

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

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

LOADING PLUGIN MODULES

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

For more information, see *Plugins*, page 341.

Starting C-SPY

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

These tasks are covered:

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

STARTING A DEBUG SESSION

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

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

LOADING EXECUTABLE FILES BUILT OUTSIDE OF THE IDE

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

To create a project for an externally built file:

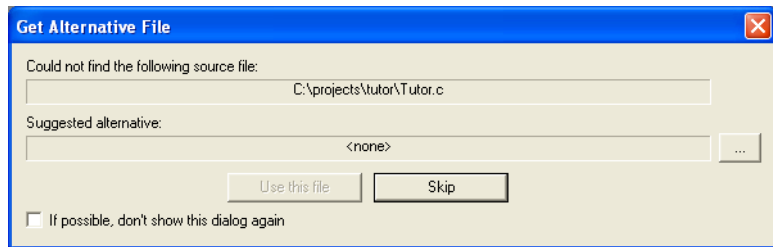
- 1 Choose **Project>Create New Project**, and specify a project name.
- 2 To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the **Files of type** drop-down list. Locate the executable file.
- ▶ 3 To start the executable file, click the **Download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

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

STARTING A DEBUG SESSION WITH SOURCE FILES MISSING

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

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



Typically, you can use the dialog box like this:

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

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

For more information, see *Get Alternative File dialog box*, page 62.

LOADING MULTIPLE IMAGES

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

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

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

To load additional images at C-SPY startup:

- 1 Choose **Project>Options>Debugger>Images** and specify up to three additional images to be loaded. For more information, see *Images*, page 340.
- 2 Start the debug session.

To load additional images at a specific moment:

Use the `__loadImage` system macro and execute it using either one of the methods described in *Using C-SPY macros*, page 253.

To display a list of loaded images:

Choose **Images** from the **View** menu. The **Images** window is displayed, see *Images window*, page 60.

EDITING IN C-SPY WINDOWS

You can edit the contents of the **Memory**, **Symbolic Memory**, **Registers**, **Register User Groups Setup**, **Auto**, **Watch**, **Locals**, **Statics**, and **Quick Watch** windows.

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

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

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

```
myArray;3
```

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

```
myPtr;3
```

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

```
myArray;5,10
```

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

```
myPtr;5,10
```

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

Adapting for target hardware

These tasks are covered:

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

MODIFYING A DEVICE DESCRIPTION FILE

C-SPY uses device description files provided with the product to handle several of the target-specific adaptations, see *Selecting a device description file*, page 51. They contain device-specific information such as:

- Memory information for device-specific memory zones, see *C-SPY memory zones*, page 134.
- Definitions of memory-mapped peripheral units, device-specific CPU registers, and groups of these.
- Definitions for device-specific interrupts, which makes it possible to simulate these interrupts in the C-SPY simulator; see *Interrupts*, page 243.

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

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

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

INITIALIZING TARGET HARDWARE BEFORE C-SPY STARTS

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

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

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

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

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

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

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

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

Reference information on starting C-SPY

Reference information about:

- *C-SPY Debugger main window*, page 56
- *Images window*, page 60
- *Get Alternative File dialog box*, page 62

See also:

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

C-SPY Debugger main window

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

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

- A special debug toolbar
- Several windows and dialog boxes specific to C-SPY.

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

Menu bar

These menus are available during a debug session:

Debug

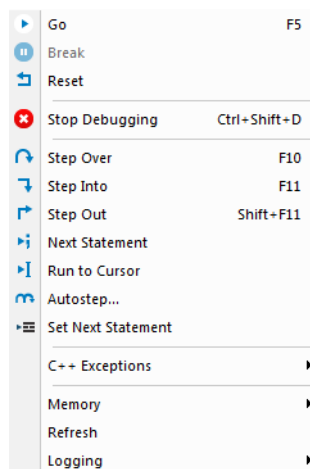
Provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

C-SPY driver menu

Provides commands specific to a C-SPY driver. The driver-specific menu is only available when the driver is used. For information about the driver-specific menu commands, see *Reference information on C-SPY driver menus*, page 367.

Debug menu

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



These commands are available:

▶ Go (F5)

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

**Break**

Stops the application execution.

**Reset**

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

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

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

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

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

**Step Over (F10)**

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

**Step Into (F11)**

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

**Step Out (Shift+F11)**

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

**Next Statement**

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

**Run to Cursor**

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

**Autostep**

Displays a dialog box where you can customize and perform autosteping, see *Autostep settings dialog box*, page 82.



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

This menu command is not supported by your product package.

C++ Exceptions>

Break on Uncaught Exception

This menu command is not supported by your product package.

Memory>Save

Displays a dialog box where you can save the contents of a specified memory area to a file, see *Memory Save dialog box*, page 143.

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 144.

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 Log file

Displays a dialog box where you can choose to log the contents of the **Debug Log** window to a file. You can select the type and the location of the log file. You can choose what you want to log: errors, warnings, system information, user messages, or all of these. See *Log File dialog box*, page 80.

Logging>

Set Terminal I/O Log file

Displays a dialog box where you can choose to log simulated target access communication to a file. You can select the destination of the log file. See *Terminal I/O Log File dialog box*, page 78

C-SPY windows

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

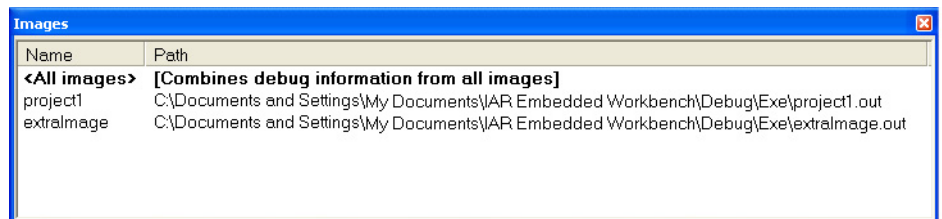
- C-SPY Debugger main window

- Disassembly window
- Memory window
- Symbolic Memory window
- Registers window
- Watch window
- Locals window
- Auto window
- Quick Watch window
- Statics window
- Call Stack window
- Trace window
- Function Trace window
- Timeline window, see *Reference information on application timeline*, page 180
- Terminal I/O window
- Code Coverage window
- Function Profiler window
- Images window
- Stack window
- Symbols window.

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

Images window

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



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

Normally, a source application consists of exactly one image that you debug. However, you can also load additional images. This means that the complete debuggable unit consists of several images. See also *Loading multiple images*, page 53.

Requirements

None; this window is always available.

Display area

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

This area lists the loaded images in these columns:

Name

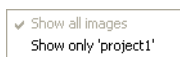
The name of the loaded image.

Path

The path to the loaded image.

Context menu

This context menu is available:



These commands are available:

Show all images

Shows debug information for all loaded debug images.

Show only *image*

Shows debug information for the selected debug image.

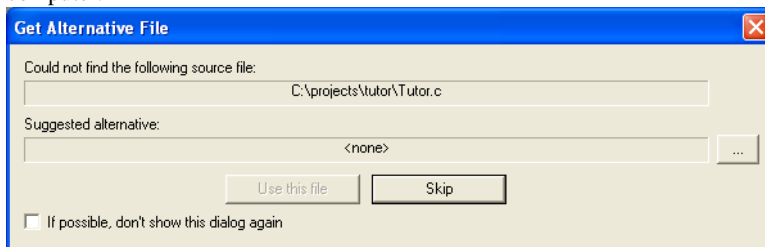
Related information

For related information, see:

- *Loading multiple images*, page 53
- *Images*, page 340
- `__loadImage`, page 275.

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 52.

Could not find the following source file

The missing source file.

Suggested alternative

Specify an alternative file.

Use this file

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

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

Skip

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

If possible, don't show this dialog again

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

Related information

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

Executing your application

- Introduction to application execution
- Reference information on application execution

Introduction to application execution

These topics are covered:

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

BRIEFLY ABOUT APPLICATION EXECUTION

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

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

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

SOURCE AND DISASSEMBLY MODE DEBUGGING

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

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

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

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

SINGLE STEPPING

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

There are several factors that can slow down the stepping speed. If you find it too slow, see *Troubleshooting slow stepping speed*, page 66 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 82.

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

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



Step Into

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

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

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



Step Over

The **Step Over** command executes to the next step point in the same function, without stopping inside called functions. The command would take you to the $g(n-2)$ function call, which is not a statement on its own but part of the same statement as $g(n-1)$. Thus, you can skip uninteresting calls which are parts of statements and instead focus on critical parts:

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



Next Statement

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

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



Step Out

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

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

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

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

TROUBLESHOOTING SLOW STEPPING SPEED

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

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

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

For more information, see *Breakpoints in the C-SPY hardware debugger drivers*, page 108 and *Breakpoint consumers*, page 110.

- 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 136.
- 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**, **Locals**, **Statics** if it is open, because all these windows read memory after each step and that might slow down stepping.
- Close the **Stack** window if it is open. Choose **Tools>Options>Stack** and disable the **Enable graphical stack display and stack usage tracking** option if it is enabled.
- If possible, increase the communication speed between C-SPY and the target board/emulator.

RUNNING THE APPLICATION

Go



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



Run to Cursor

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

HIGHLIGHTING

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

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

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

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

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 *AVR® IAR Assembler Reference Guide*.

TERMINAL INPUT AND OUTPUT

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



This facility is useful in two different contexts:

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

For more information, see *Terminal I/O window*, page 77 and *Terminal I/O Log File dialog box*, page 78.

DEBUG LOGGING

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



It can sometimes be convenient to log the information to a file where you can easily inspect it, see *Log File dialog box*, page 80. The two main advantages are:

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

Reference information on application execution

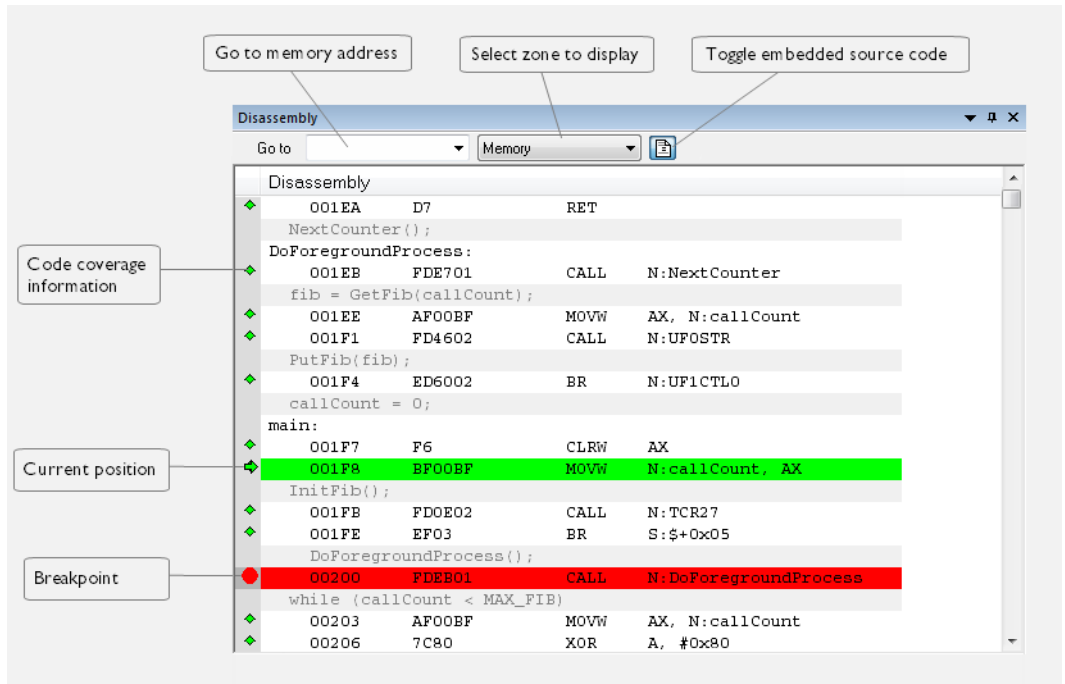
Reference information about:

- *Disassembly window*, page 71
- *Call Stack window*, page 75
- *Terminal I/O window*, page 77
- *Terminal I/O Log File dialog box*, page 78
- *Debug Log window*, page 79
- *Log File dialog box*, page 80
- *Report Assert dialog box*, page 81
- *Autostep settings dialog box*, page 82
- *Cores window*, page 82

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

Disassembly window

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



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

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

- 1 Choose **Tools>Options>Debugger**.
- 2 Set the default color using the **Source code coloring in disassembly window** option.



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

See also *Source and disassembly mode debugging*, page 63.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 134.

Toggle Mixed-Mode

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

Display area

The display area shows the disassembled application code.

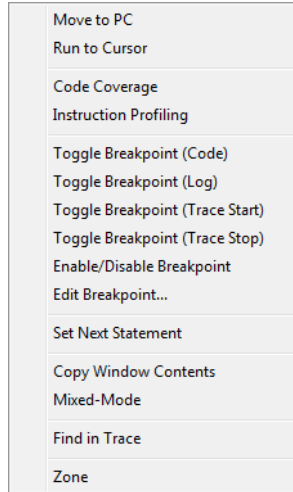
This area contains these graphic elements:

Green highlight	Indicates the current position, that is the next assembler instruction to be executed. To move the cursor to any line in the Disassembly window, click the line. Alternatively, move the cursor using the navigation keys.
Yellow highlight	Indicates a position other than the current position, such as when navigating between frames in the Call Stack window or between items in the Trace window.
Red dot	Indicates a breakpoint. Double-click in the gray left-side margin of the window to set a breakpoint. For more information, see <i>Breakpoints</i> , page 105.
Green diamond	Indicates code that has been executed—that is, code coverage.

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

Context menu

This context menu is available:



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

These commands are available:

Move to PC

Displays code at the current program counter location.

Run to Cursor

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

Code Coverage

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

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

Instruction Profiling

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

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

Toggle Breakpoint (Code)

Toggles a code breakpoint. Assembler instructions and any corresponding label at which code breakpoints have been set are highlighted in red. For more information, see *Code breakpoints dialog box*, page 120.

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 121.

Toggle Breakpoint (Trace Start)

Toggles a Trace Start breakpoint. When the breakpoint is triggered, the trace data collection starts. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Start breakpoints dialog box*, page 168.

Toggle Breakpoint (Trace Stop)

Toggles a Trace Stop breakpoint. When the breakpoint is triggered, the trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports trace. For more information, see *Trace Stop breakpoints dialog box*, page 169.

Enable/Disable Breakpoint

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

Edit Breakpoint

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

Set Next Statement

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

Copy Window Contents

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

Mixed-Mode

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

Find in Trace

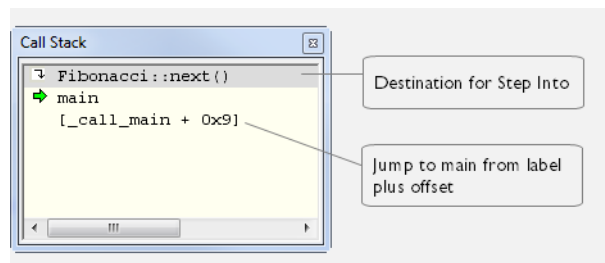
Searches the contents of the **Trace** window for occurrences of the given location—the position of the insertion point in the source code—and reports the result in the **Find in Trace** window. This menu command requires support for Trace in the C-SPY driver you are using, see *Differences between the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

Zone

Selects a memory zone, see *C-SPY memory zones*, page 134.

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 68.

Requirements

None; this window is always available.

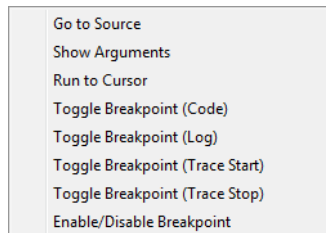
Display area

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

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

Context menu

This context menu is available:



These commands are available:

Go to Source

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

Show Arguments

Shows function arguments.

Run to Cursor

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

Toggle Breakpoint (Code)

Toggles a code breakpoint.

Toggle Breakpoint (Log)

Toggles a log breakpoint.

Toggle Breakpoint (Trace Start)

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

Toggle Breakpoint (Trace Stop)

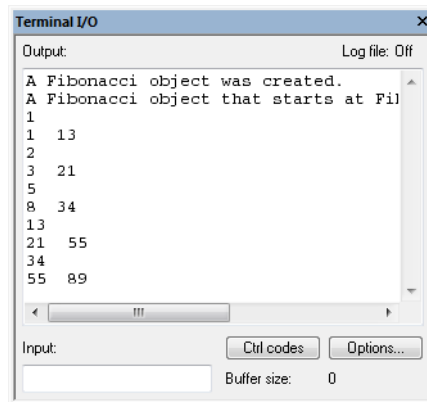
Toggles a Trace Stop breakpoint. When the breakpoint is triggered, trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports it.

Enable/Disable Breakpoint

Enables or disables the selected breakpoint

Terminal I/O window

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



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

To use this window, you must:

- I Link your application with the option **With I/O emulation modules**.

C-SPY will then direct `stdin`, `stdout` and `stderr` to this window. If the **Terminal I/O** window is closed, C-SPY will open it automatically when input is required, but not for output.

See also *Terminal input and output*, page 69.

Requirements

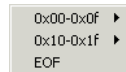
None; this window is always available.

Input

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

Ctrl codes

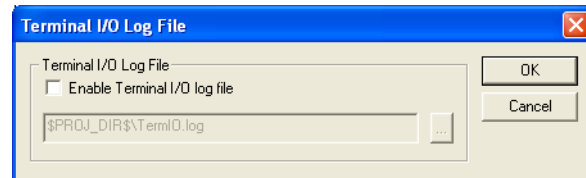
Opens a menu for input of special characters, such as EOF (end of file) and NUL.

**Options**

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

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 69.

Requirements

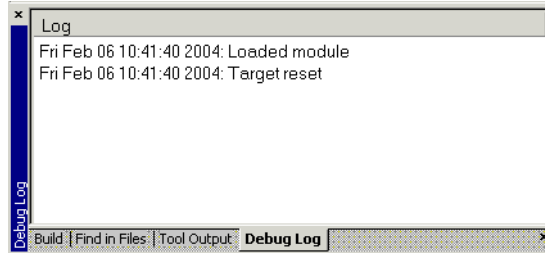
None; this dialog box is always available.

Terminal IO Log Files

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

Debug Log window

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



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

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>
```

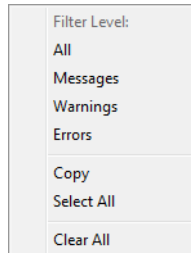
See also *Debug logging*, page 69 and *Log File dialog box*, page 80.

Requirements

None; this window is always available.

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

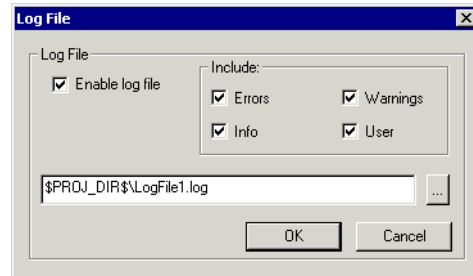
Selects the contents of the window.

Clear All

Clears the contents of the window.

Log File dialog box

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



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

Requirements

None; this dialog box is always available.

Enable Log file

Enables or disables logging to the file.

Include

The information printed in the file is, by default, the same as the information listed in the Log window. Use the browse button, to override the default file and location of the

log file (the default filename extension is `log`). To change the information logged, choose between:

Errors

C-SPY has failed to perform an operation.

Warnings

An error or omission of concern.

Info

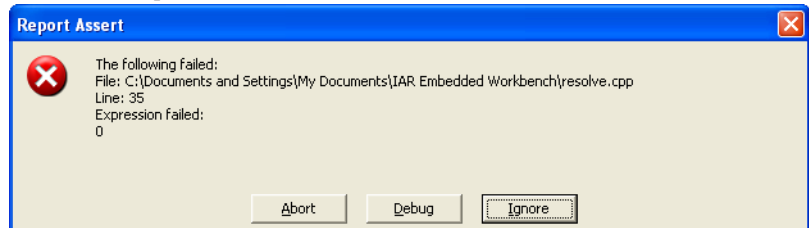
Progress information about actions C-SPY has performed.

User

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

Report Assert dialog box

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



Abort

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

Debug

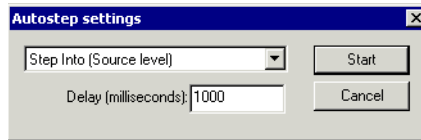
C-SPY stops the execution of the application and returns control to you.

Ignore

The assertion is ignored and the application continues to execute.

Autostep settings dialog box

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



Use this dialog box to customize autostepping.

The drop-down menu lists the available step commands, see *Single stepping*, page 64.

Requirements


None; this dialog box is always available.

Delay

Specify the delay between each step in milliseconds.

Cores window

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

Cores				
Core	Status	PC	Cycles	
 0: Core 0	Stopped	0xE055F	74	

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

Requirements

None; this window is always available.

Display area






A row in this area shows information about a core, in these columns:

Execution state

Displays one of these icons to indicate the execution state of the core.



in focus, not executing

	not in focus, not executing
	in focus, executing
	not in focus, executing
	in focus, in sleep mode
	not in focus, in sleep mode

Core

The name of the core.

Status

The status of the execution, which can be one of **Stopped**, **Running**, or **Sleeping**.

PC

The value of the program counter.

Cycles | Time

The value of the cycle counter or the execution time since the start of the execution, depending on the debugger driver you are using.

Variables and expressions

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

Introduction to working with variables and expressions

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

These topics are covered:

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

BRIEFLY ABOUT WORKING WITH VARIABLES AND EXPRESSIONS

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

- **Tooltip watch**—in the editor window—provides the simplest way of viewing the value of a variable or more complex expressions. Just point at the variable with the mouse pointer. The value is displayed next to the variable.
- The **Auto** window displays a useful selection of variables and expressions in, or near, the current statement. The window is automatically updated when execution stops.
- The **Locals** window displays the local variables, that is, auto variables and function parameters for the active function. The window is automatically updated when execution stops.
- The **Watch** window allows you to monitor the values of C-SPY expressions and variables. The window is automatically updated when execution stops.
- The **Statics** window displays the values of variables with static storage duration. The window is automatically updated when execution stops.
- The **Macro Quicklaunch** window and the **Quick Watch** window give you precise control over when to evaluate an expression.

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

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

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

For more information about these windows, see *The application timeline*, page 175.

C-SPY EXPRESSIONS

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

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

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

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

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

C/C++ symbols

C symbols are symbols that you have defined in the C source code of your application, for instance variables, constants, and functions (functions can be used as symbols but cannot be executed). C symbols can be referenced by their names. Note that C++ symbols might implicitly contain function calls which are not allowed in C-SPY symbols and expressions.

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

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

However, this line will be accepted:

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

Assembler symbols

Assembler symbols can be assembler labels or registers, for example the program counter, the stack pointer, or other CPU registers. If a device description file is used, all memory-mapped peripheral units, such as I/O ports, can also be used as assembler symbols in the same way as the CPU registers. See *Modifying a device description file*, page 55.

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

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

Table 4: C-SPY assembler symbols expressions

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

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

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

Which processor-specific symbols are available by default can be seen in the **Registers** window, using the CPU Registers register group. See *Registers window*, page 153.

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 252.

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 258.

Using sizeof

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

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

The former is for types and the latter for expressions.

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

LIMITATIONS ON VARIABLE INFORMATION

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

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

Effects of optimizations

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

Consider this example:

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


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

```
Unavailable
```

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

Working with variables and expressions

These tasks are covered:

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

See also *Analyzing your application's timeline*, page 177.

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.

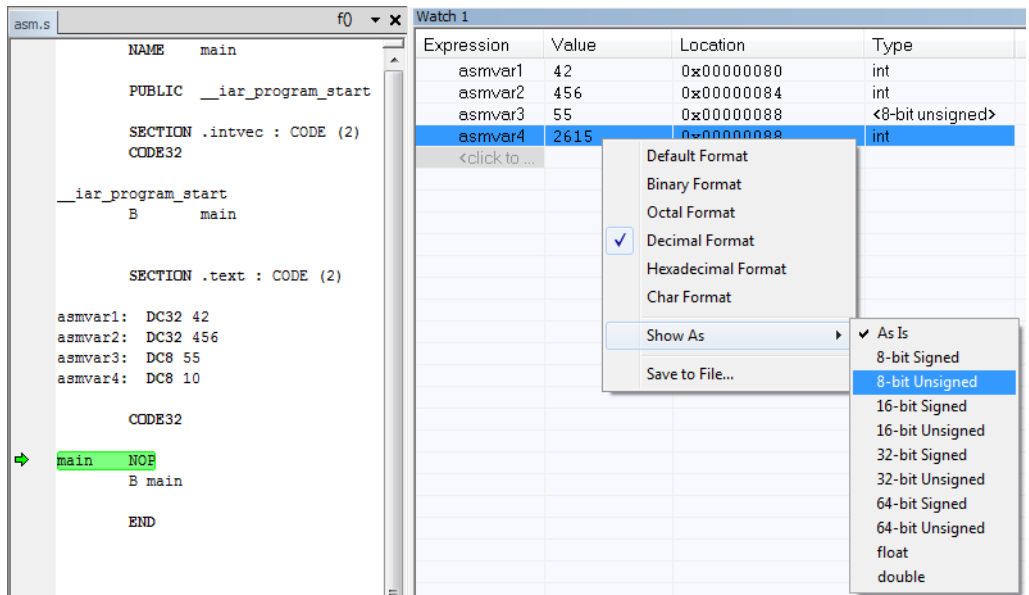
Right-click in any of the windows to access the context menu which contains additional commands. Convenient drag-and-drop between windows is supported, except for in the **Locals** window, Data logging windows, and the **Quick Watch** window where it is not relevant.

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

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



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

Reference information on working with variables and expressions

Reference information about:

- *Auto window*, page 91
- *Locals window*, page 93
- *Watch window*, page 95
- *Statics window*, page 97
- *Quick Watch window*, page 100
- *Symbols window*, page 102

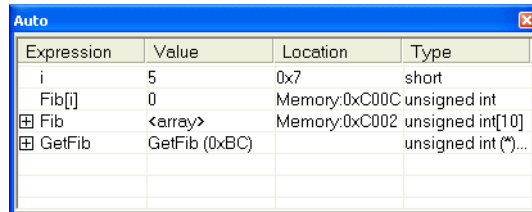
- *Resolve Symbol Ambiguity dialog box*, page 104

See also:

- *Reference information on trace*, page 164 for trace-related reference information
- *Macro Quicklaunch window*, page 306

Auto window

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



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

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

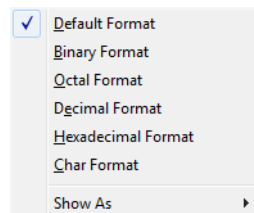
See also *Editing in C-SPY windows*, page 54.

Requirements

None; this window is always available.

Context menu

This context menu is available:



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

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

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

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

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

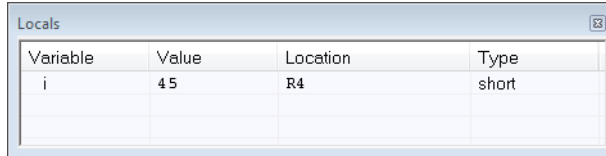
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 89.

Locals window

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



Variable	Value	Location	Type
i	45	R4	short

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

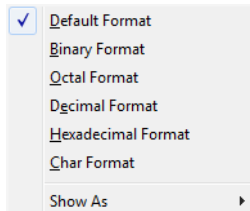
See also *Editing in C-SPY windows*, page 54.

Requirements

None; this window is always available.

Context menu

This context menu is available:



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

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 89.

Watch window

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

Expression	Value	Location	Type
callCount	2	Memory : 0xFBFB00	int
Fib	<array>	Memory : 0xFBFB02	unsigned int[...]
[0]	1	Memory : 0xFBFB02	unsigned int
[1]	1	Memory : 0xFBFB04	unsigned int
[2]	2	Memory : 0xFBFB06	unsigned int
[3]	3	Memory : 0xFBFB08	unsigned int
[4]	5	Memory : 0xFBFB0A	unsigned int
[5]	8	Memory : 0xFBFB0C	unsigned int
[6]	13	Memory : 0xFBFB0E	unsigned int
[7]	21	Memory : 0xFBFB10	unsigned int
[8]	34	Memory : 0xFBFB12	unsigned int
[9]	55	Memory : 0xFBFB14	unsigned int
<click to ...>			

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

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



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

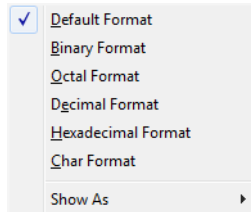
See also *Editing in C-SPY windows*, page 54.

Requirements

None; this window is always available.

Context menu

This context menu is available:



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

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 89.

Statics window

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

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

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

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

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

See also *Editing in C-SPY windows*, page 54.

To select variables to monitor:

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

Requirements

None; this window is always available.

Display area

This area contains these columns:

Expression

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

Value

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

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

This column is editable.

Location

The location in memory where this variable is stored.

Type

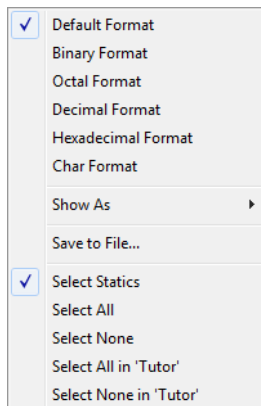
The data type of the variable.

Module

The module of the variable.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Save to File

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

Select Statics

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

Select All

Selects all variables.

Select None

Deselects all variables.

Select All in *module*

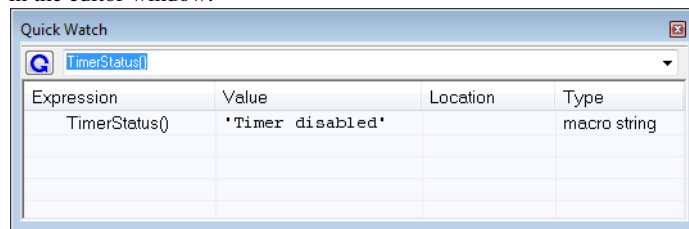
Selects all variables in the selected module.

Select None in *module*

Deselects all variables in the selected module.

Quick Watch window

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



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

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

See also *Editing in C-SPY windows*, page 54.

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.

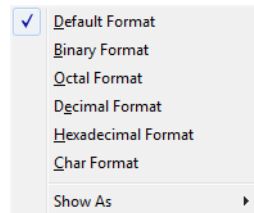
For an example, see *Using C-SPY macros*, page 253.

Requirements

None; this window is always available.

Context menu

This context menu is available:



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

These commands are available:

Remove

Removes the selected expression from the window.

Remove All

Removes all expressions listed in the window.

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

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

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

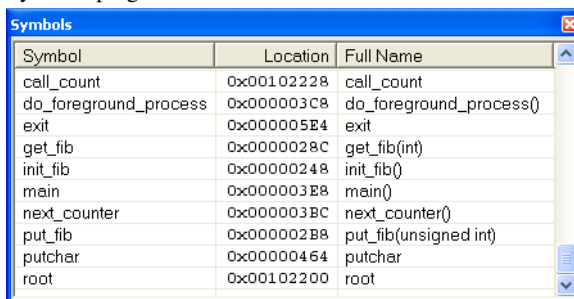
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 89.

Symbols window

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



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

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

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

Requirements

None; this window is always available.

Display area

This area contains these columns:

Symbol

The symbol name.

Location

The memory address.

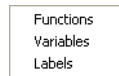
Full name

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

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

Context menu

This context menu is available:



These commands are available:

Functions

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

Variables

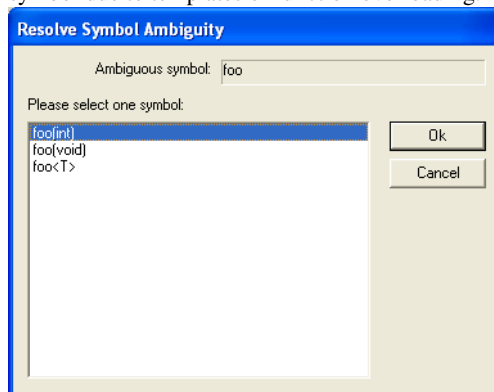
Toggles the display of variables on or off in the list.

Labels

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

Resolve Symbol Ambiguity dialog box

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



Requirements

None; this window is always available.

Ambiguous symbol

Indicates which symbol that is ambiguous.

Please select one symbol

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

Breakpoints

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

Introduction to setting and using breakpoints

These topics are covered:

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

REASONS FOR USING BREAKPOINTS

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

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

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

BRIEFLY ABOUT SETTING BREAKPOINTS

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

appear in the Breakpoints window. From this window you can conveniently view all breakpoints, enable and disable breakpoints, and open a dialog box for defining new breakpoints. The **Breakpoint Usage** window also lists all internally used breakpoints, see *Breakpoint consumers*, page 110.

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 64.

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

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

BREAKPOINT TYPES

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

Code breakpoints

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

Log breakpoints

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

Trace Start and Stop breakpoints

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

Data breakpoints

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

small parts of the code. Data breakpoints are triggered when data is accessed at the specified location. The execution will usually stop directly after the instruction that accessed the data has been executed.

Data Log breakpoints

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

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

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

Immediate breakpoints

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

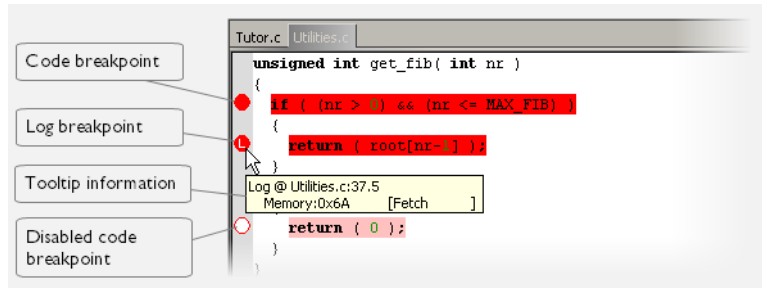
This type of breakpoint is useful for simulating memory-mapped devices of various kinds (for instance serial ports and timers). When the simulated processor reads from a memory-mapped location, a C-SPY macro function can intervene and supply appropriate data. Conversely, when the simulated processor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

Complex breakpoints

The C-SPY Atmel-ICE driver, the C-SPY Power Debugger driver, the C-SPY AVR ONE! driver, and the C-SPY JTAGICE3 driver support complex breakpoints. Complex breakpoints use the functionality of the firmware and are faster than data breakpoints and code breakpoints. Using complex breakpoints, you can specify special conditions for when the breakpoint should trigger.

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 AVR*.



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

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

BREAKPOINTS IN THE C-SPY SIMULATOR

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

BREAKPOINTS IN THE C-SPY HARDWARE DEBUGGER DRIVERS

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

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

C-SPY hardware debugger driver	Code breakpoints	Data breakpoints
JTAGICE		
using hardware breakpoints ⁸	4 ¹	2 ¹
using software breakpoints	Unlimited	2 ¹
JTAGICE mkII		
using hardware breakpoints ^{3,8}	4 ^{1,2}	2 ^{1,4}
using software breakpoints	Unlimited	2 ^{1,4}
Atmel-ICE		
using hardware breakpoints ^{3,5,7, 8}	4 ^{1,2}	2 ⁶
using software breakpoints	Unlimited	2 ⁶
Power Debugger		
using hardware breakpoints ^{3,5,7, 8}	4 ^{1,2}	2 ⁶
using software breakpoints	Unlimited	2 ⁶
JTAGICE3		
using hardware breakpoints ^{3,5,7, 8}	4 ^{1,2}	2 ⁶
using software breakpoints	Unlimited	2 ⁶
AVR ONE!		
using hardware breakpoints ^{3,5,7, 8}	4 ^{1,2}	2 ⁶
using software breakpoints	Unlimited	2 ⁶

Table 6: Available breakpoints in C-SPY hardware debugger drivers

¹ The sum of code and data breakpoints can never exceed 4—the number of available hardware breakpoints. This means that for every data breakpoint in use, one less code breakpoint is available, and that no data breakpoints are available if you use four code breakpoints.

² If software breakpoints are enabled, the number of code breakpoints is unlimited.

³ When the number of available hardware breakpoints is exceeded, software breakpoints will be used if enabled.

⁴ Data breakpoints are not available when the debugWIRE interface is used.

⁵ If data breakpoints and complex breakpoints have not been used, hardware breakpoints will be used until exhausted. After that, software breakpoints will be used.

⁶ If complex breakpoints are used, data breakpoints are not available, and vice versa.

⁷ Note that a complex breakpoint uses all available hardware breakpoints.

⁸ The number of available hardware breakpoints depends on the target system you are using.

For Atmel-ICE, Power Debugger, JTAGICE3, JTAGICE mkII, AVRONE!, and Dragon, the number and types of breakpoints available depend on whether the device is using the JTAG or the debugWIRE interface. The information in this guide reflects the JTAG interface. When a device with debugWIRE is used, data breakpoints are not available and the debugger will use software code breakpoints.

If the driver and the device support software breakpoints and they 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.

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 **With I/O emulation modules** has been selected.
 - In the DLIB runtime environment, C-SPY will set a system breakpoint on the `__DebugBreak` label.
 - In the CLIB runtime environment, C-SPY will set a breakpoint if:
 - the library functions `putchar` and `getchar` are used (low-level routines used by functions like `printf` and `scanf`)
 - the application has an `exit` label.

You can disable the setting of system breakpoints on the `putchar` and `getchar` functions and on the `exit` label; see .

For more information about the option **System breakpoints on**:

- For AVRONE!, see *AVR ONE! 2*, page 350.
- For Atmel-ICE, see *Atmel-ICE 2*, page 345.
- For JTAGICE3, see *JTAGICE3 2*, page 353.
- For JTAGICE mkII, see *JTAGICE mkII 2*, page 357.
- For Dragon, see *Dragon 2*, page 361.
- For Power Debugger. see *Power Debugger 2*, page 364.

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

C-SPY plugin modules

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

To disable the breakpoint used by the Stack window:

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

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

Setting breakpoints

These tasks are covered:

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

VARIOUS WAYS TO SET A BREAKPOINT

You can set a breakpoint in various ways:

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

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

TOGGLING A SIMPLE CODE BREAKPOINT

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



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

SETTING BREAKPOINTS USING THE DIALOG BOX

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

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

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

To set a new breakpoint:

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

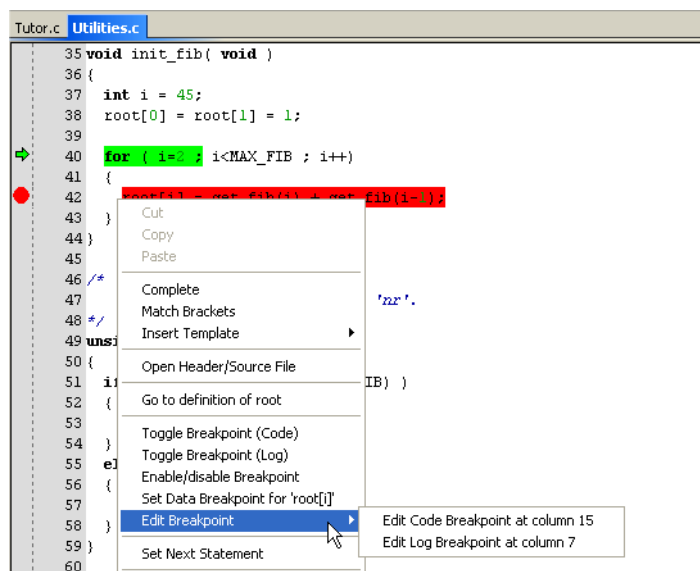
Depending on the C-SPY driver you are using, different breakpoint types are available.

- 4 In the breakpoint dialog box that appears, specify the breakpoint settings and click **OK**.

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

To modify an existing breakpoint:

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



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

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

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

SETTING A DATA BREAKPOINT IN THE MEMORY WINDOW

You can set breakpoints directly on a memory location in the **Memory** window.

Right-click in the window and choose the breakpoint command from the context menu that appears. To set the breakpoint on a range, select a portion of the memory contents.

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

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

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

SETTING BREAKPOINTS USING SYSTEM MACROS

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

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

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

These breakpoint macros are available:

C-SPY macro for breakpoints	Simulator	Atmel-ICE	Power Debugger	JTAGICE3	JTAGICE mkII	AVR ONE!
__setCodeBreak	Yes	Yes	Yes	Yes	Yes	Yes
__setDataBreak	Yes	Yes	Yes	Yes	—	Yes
__setLogBreak	Yes	Yes	Yes	Yes	Yes	Yes
__setDataLogBreak	Yes	—	—	—	—	—
__setSimBreak	Yes	—	—	—	—	—
__setTraceStartBreak	Yes	—	—	—	—	—
__setTraceStopBreak	Yes	—	—	—	—	—
__clearBreak	Yes	Yes	Yes	Yes	Yes	Yes

Table 7: C-SPY macros for breakpoints

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 265.

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 253.

USEFUL BREAKPOINT HINTS

Below are some useful hints related to setting breakpoints.



Tracing incorrect function arguments

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

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

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

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

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

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

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



Performing a task and continuing execution

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

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

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

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

```

__var my_counter;

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

```

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

Reference information on breakpoints

Reference information about:

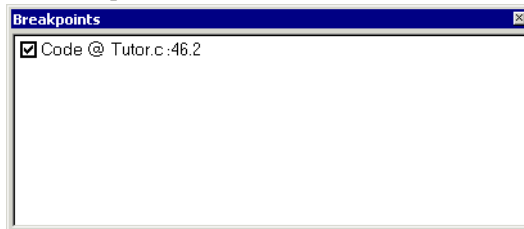
- *Breakpoints window*, page 117
- *Breakpoint Usage window*, page 119
- *Code breakpoints dialog box*, page 120
- *Log breakpoints dialog box*, page 121
- *Data breakpoints dialog box*, page 123
- *Data Log breakpoints dialog box*, page 125
- *Immediate breakpoints dialog box*, page 126
- *Complex breakpoints dialog box*, page 127
- *Enter Location dialog box*, page 130
- *Resolve Source Ambiguity dialog box*, page 131.

See also:

- *Reference information on C-SPY system macros*, page 265
- *Reference information on trace*, page 164.

Breakpoints window

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



This window lists all breakpoints you define.

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

Requirements

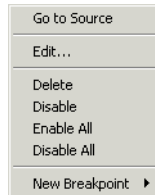
None; this window is always available.

Display area

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

Context menu

This context menu is available:



These commands are available:

Go to Source

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

Edit

Opens the breakpoint dialog box for the breakpoint you selected.

Delete

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

Enable

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

Disable

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

Enable All

Enables all defined breakpoints.

Disable All

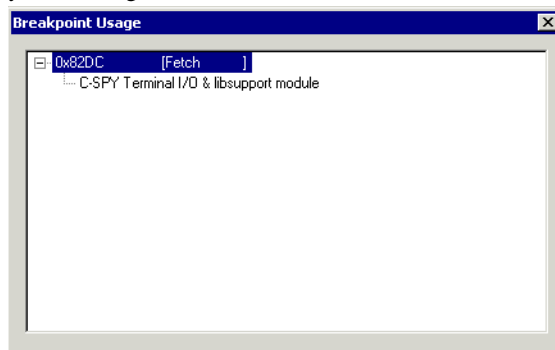
Disables all defined breakpoints.

New Breakpoint

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

Breakpoint Usage window

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



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

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

C-SPY uses breakpoints when stepping, see the **Breakpoint Usage** window for:

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

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

Requirements

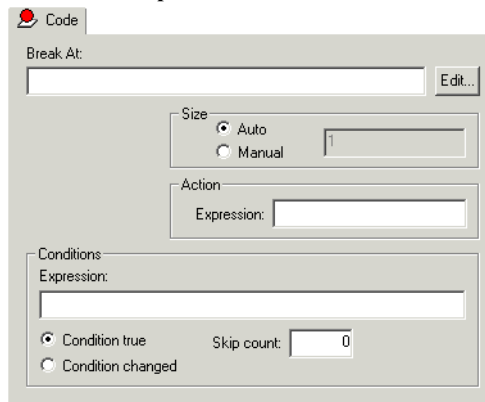
None; this window is always available.

Display area

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

Code breakpoints dialog box

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



This figure reflects the C-SPY simulator.

Use the **Code** breakpoints dialog box to set a code breakpoint, see *Setting breakpoints using the dialog box*, page 112.

Requirements

None; this dialog box is always available.

Break At

Specify the code location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 130.

Size

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

Auto

The size will be set automatically, typically to 1.

Manual

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

Action

Specify a valid C-SPY expression, which is evaluated when the breakpoint is triggered and the condition is true. For more information, see *Useful breakpoint hints*, page 115.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 86.

Condition true

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

Condition changed

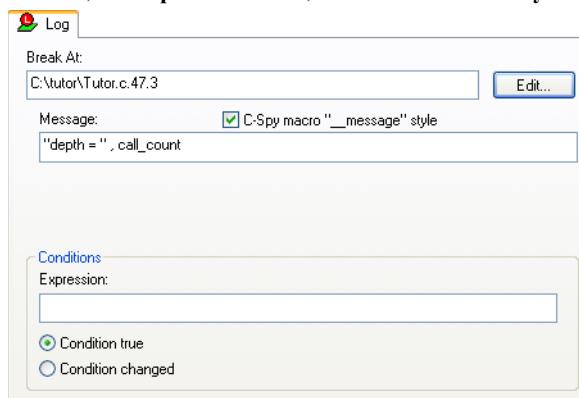
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Skip count

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

Log breakpoints dialog box

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



This figure reflects the C-SPY simulator.

Use the **Log** breakpoints dialog box to set a log breakpoint, see *Setting breakpoints using the dialog box*, page 112.

Requirements

None; this dialog box is always available.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 130.

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 261.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 86.

Condition true

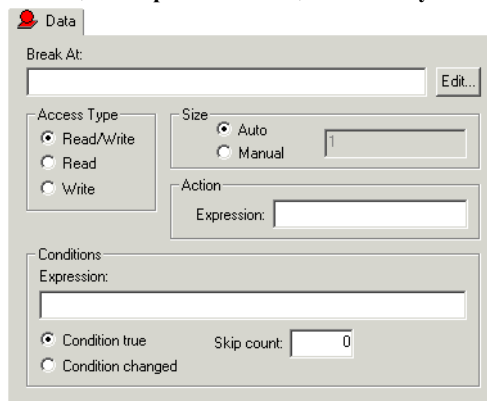
The breakpoint is triggered if the value of the expression is true.

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 112. 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 Atmel ICE driver
- The C-SPY Power Debugger driver
- The C-SPY JTAGICE3 driver
- The C-SPY AVR ONE! driver
- The C-SPY JTAGICE mkII driver, unless the debugWIRE interface is used
- The C-SPY Dragon driver.

Break At

Specify the data location of the breakpoint in the text box. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 130.

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 115.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 86.

Condition true

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

Condition changed

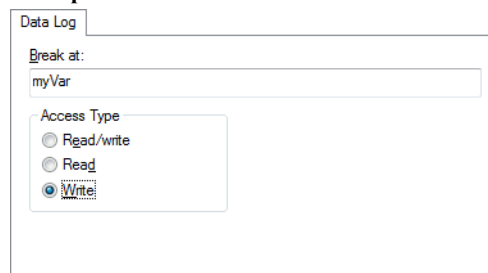
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

Skip count

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

Data 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 112.

See also *Data Log breakpoints*, page 107 and *Getting started using data logging*, page 179.

Requirements

The C-SPY simulator.

Trigger at

Specify the data location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 130.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read/Write

Reads from or writes to location.

Read

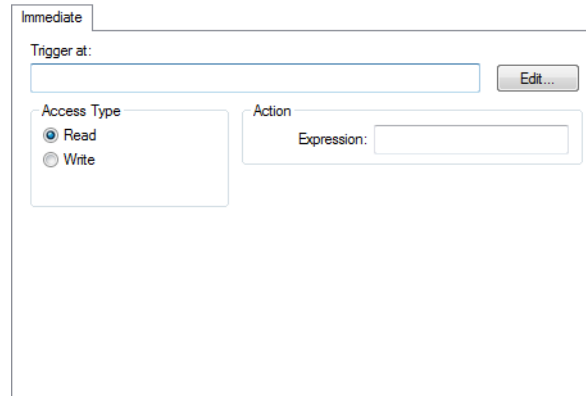
Reads from location.

Write

Writes to location.

Immediate breakpoints dialog box

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



In the C-SPY simulator, use the **Immediate** breakpoints dialog box to set an immediate breakpoint, see *Setting breakpoints using the dialog box*, page 112. Immediate breakpoints do not stop execution at all; they only suspend it temporarily.

Requirements

The C-SPY simulator.

Trigger at

Specify the data location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 130.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read

Reads from location.

Write

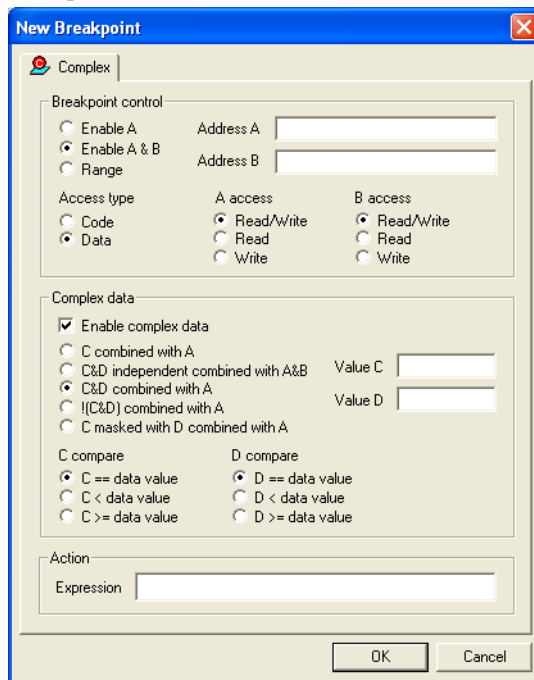
Writes to location.

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 115.

Complex breakpoints dialog box

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



Use this dialog box to set a complex breakpoint. Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoint** window and choose **Edit** on the context menu.

Complex breakpoints use the functionality of the firmware and are faster than data breakpoints and code breakpoints.

Note: A complex breakpoint uses all available hardware breakpoints.

Requirements

One of these alternatives:

- The C-SPY Atmel-ICE driver
- The C-SPY Power Debugger driver
- The C-SPY JTAGICE3 driver
- The C-SPY AVR ONE! driver.

Breakpoint control

Controls what access type at the specified address that causes a break. Choose between:

Enable A	Breaks at the address specified in the Address A text box.
Enable A&B	Breaks both at the address specified in Address A and at the address in Address B .
Range	Breaks when an address from Address A up to and including Address B is accessed. This can be useful if you want the breakpoint to be triggered on access to data structures, such as arrays, structs, and unions. When using Range , only A access is available.
A access/B access	Specifies the type of memory access that triggers complex breakpoints. Read/Write , Reads from or writes to location Read , Reads from location Write , Writes to location

Address A/B

Specify the code or data addresses where you want to set a breakpoint.

Access type

Selects the memory space, **Code** or **Data**, for the addresses in **Address A** and **Address B**. Note that both addresses must have the same access type.

Complex data

Enable complex data enables the data compare functionality.

Value C/D

Specify 1-byte numbers for the compare functionality.

C combined with A	Breaks when Address A is accessed using A access and Value C matches the memory contents at Address A according to C compare .
C&D independent combined with A&B	Breaks when Address A is accessed using A access and Value C matches the memory contents at Address A according to C compare <i>or</i> when Address B is accessed using B access and Value D matches the memory contents at Address B according to D compare .
C&D combined with A	Breaks when Address A is accessed using A access and Value C matches the memory contents at Address A according to C compare and Value D matches the memory contents at Address A according to D compare .
(C&D) combined with A	Breaks when Address A is accessed using A access and Value C does not match the memory contents of Address A according to C compare and/or Value D does not match the memory contents of Address A according to D compare .
C masked with D combined with A	Breaks when Address A is accessed using A access and Value C masked with Value D matches the memory contents at Address A according to C compare .

C/D compare

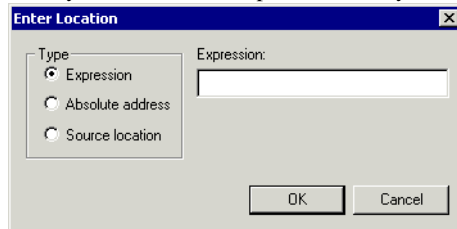
Specify the relationship between **Value C** or **Value D** and the contents of data memory at **Address A** and **Address B**.

Action

Specify an expression, for instance a C-SPY macro function, which is evaluated when the breakpoint is triggered and the condition is true.

Enter Location dialog box

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



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

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

Type

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

Expression

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

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

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

For more information about C-SPY expressions, see *C-SPY expressions*, page 86.

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 134.

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.

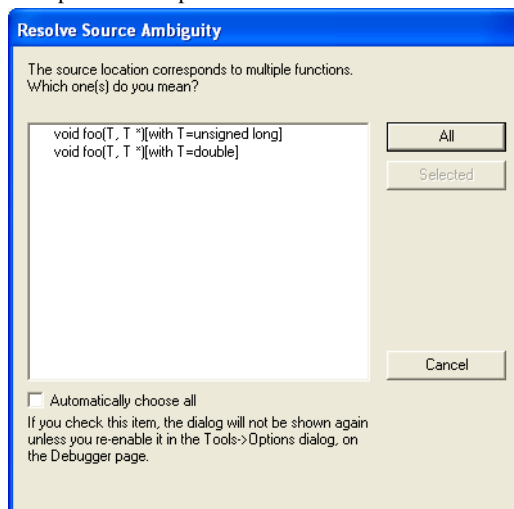
For example, `{C:\src\prog.c}.22.3`

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 AVR*.

Memory and registers

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

Introduction to monitoring memory and registers

These topics are covered:

- Briefly about monitoring memory and registers
- C-SPY memory zones

BRIEFLY ABOUT MONITORING MEMORY AND REGISTERS

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

- The **Memory** window
Gives an up-to-date display of a specified area of memory—a memory zone—and allows you to edit it. *Data coverage* along with execution of your application is highlighted with different colors. You can fill specified areas with specific values and you can set breakpoints directly on a memory location or range. You can open several instances of this window, to monitor different memory areas. The content of the window can be regularly updated while your application is executing.
- The **Symbolic Memory** window
Displays how variables with static storage duration are laid out in memory. This can be useful for better understanding memory usage or for investigating problems caused by variables being overwritten, for example by buffer overruns.
- The **Stack** window
Displays the contents of the stack, including how stack variables are laid out in memory. In addition, integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the **Stack** window is useful for determining the optimal size of the stack. You can open up to two instances of this window, each showing different stacks or different display modes of the same stack.

- The **Registers** window

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

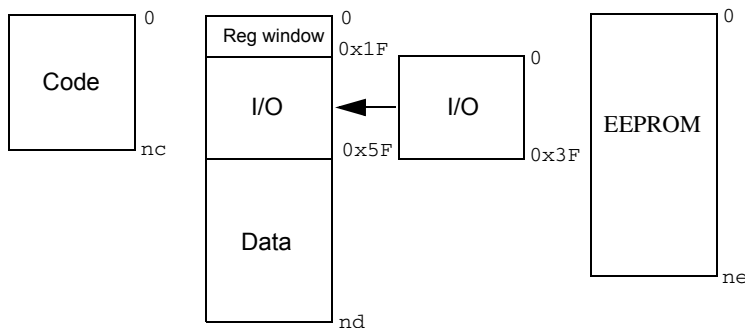
To view the memory contents for a specific variable, simply drag the variable to the **Memory** window or the **Symbolic** memory window. The memory area where the variable is located will appear.



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

C-SPY MEMORY ZONES

In C-SPY, the term *zone* is used for a named memory area. A memory address, or *location*, is a combination of a zone and a numerical offset into that zone. By default, four address zones—CODE, DATA, EEPROM, and IO_SPACE—in the debugger cover the whole AVR memory range.



nc to ne indicate the size of the memories

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

Device-specific zones

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

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

If your hardware does not have the same memory layout as any of the predefined device description files, you can define customized zones by adding them to the file.

For more information, see *Selecting a device description file*, page 51 and *Modifying a device description file*, page 55.

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).

Monitoring memory and registers

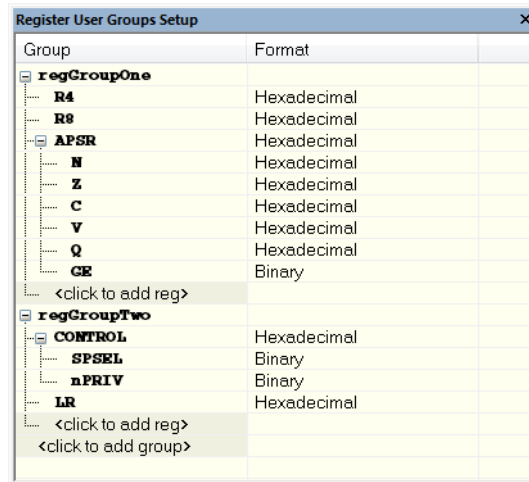
These tasks are covered:

- *Defining application-specific register groups*, page 136
- *Monitoring stack usage*, page 136

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.

- 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 156.

- 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.

MONITORING STACK USAGE

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

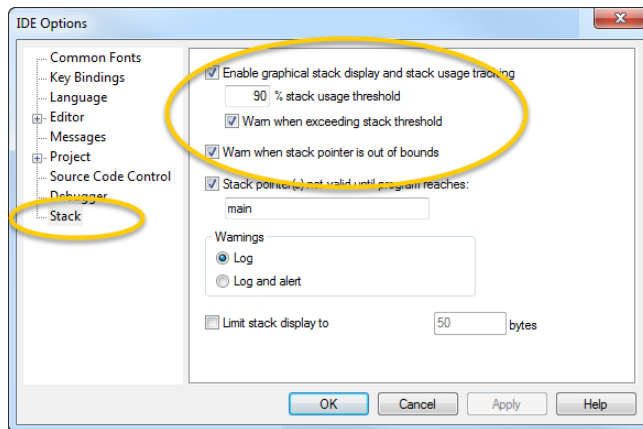
- Monitoring stack memory usage

- 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 segment holding the stack. If you, for some reason, have modified the stack initialization in the system startup code, `cstartup`, you should also change the segment definition in the linker configuration file accordingly; otherwise the **Stack** window cannot track the stack usage. For more information about this, see the *IAR C/C++ Compiler Reference Guide for AVR*.

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.
 - Notice 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.

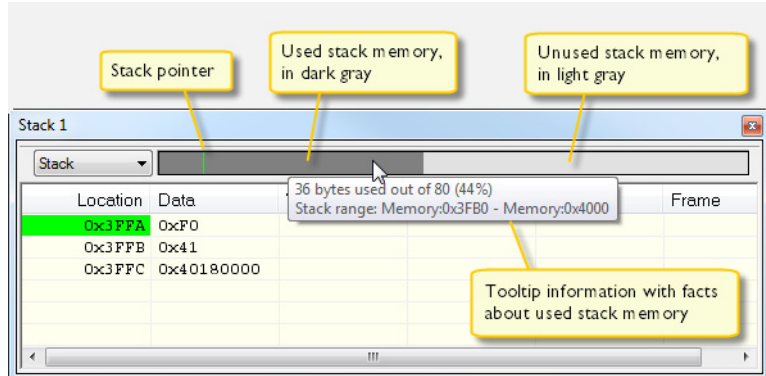
- 3 Choose **View>Stack>Stack 1** to open the **Stack** window.

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

- 4 Start executing your application.

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

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



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

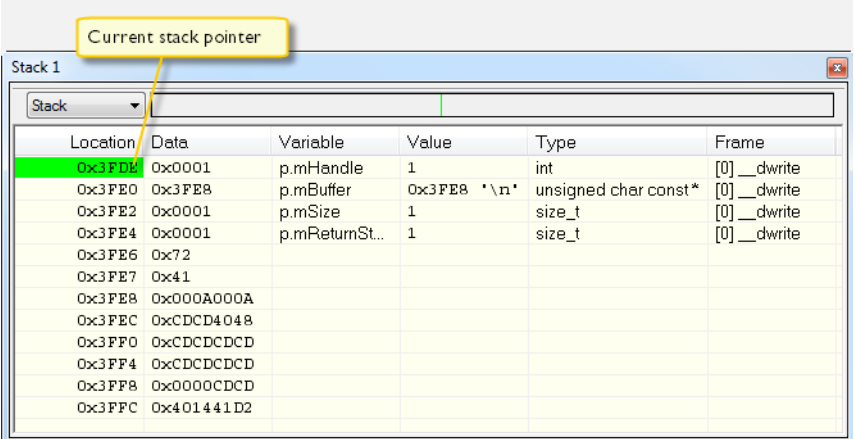
To monitor the stack memory content:

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

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

- 4 Start executing your application.

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



Location	Data	Variable	Value	Type	Frame
0x3FD4	0x0001	p.mHandle	1	int	[0] __dwrite
0x3FE0	0x3FE8	p.mBuffer	0x3FE8 '\n'	unsigned char const*	[0] __dwrite
0x3FE2	0x0001	p.mSize	1	size_t	[0] __dwrite
0x3FE4	0x0001	p.mReturnSt...	1	size_t	[0] __dwrite
0x3FE6	0x72				
0x3FE7	0x41				
0x3FE8	0x000A000A				
0x3FEC	0xCDCD4048				
0x3FF0	0xCDCDCDCD				
0x3FF4	0xCDCDCDCD				
0x3FF8	0x0000CDCD				
0x3FFC	0x401441D2				

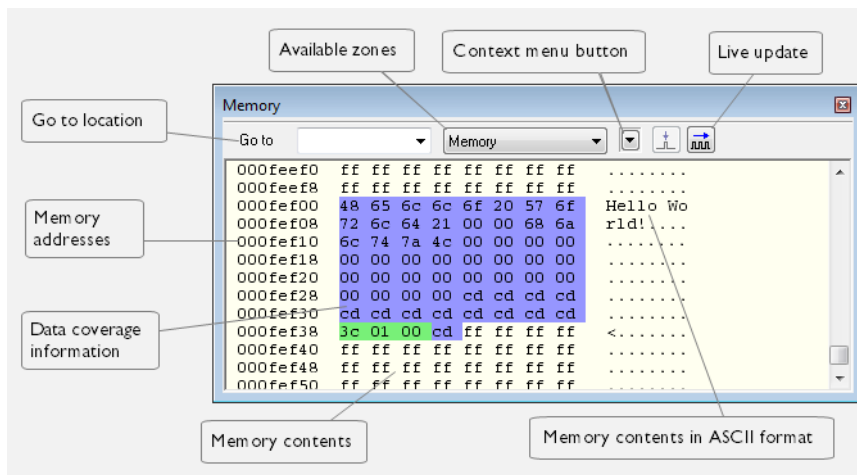
Reference information on memory and registers

Reference information about:

- *Memory window*, page 140
- *Memory Save dialog box*, page 143
- *Memory Restore dialog box*, page 144
- *Fill dialog box*, page 145
- *Symbolic Memory window*, page 146
- *Stack window*, page 149
- *Registers window*, page 153
- *Register User Groups Setup window*, page 156

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 54.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 134.

Context menu button

Displays the context menu.

Display area

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

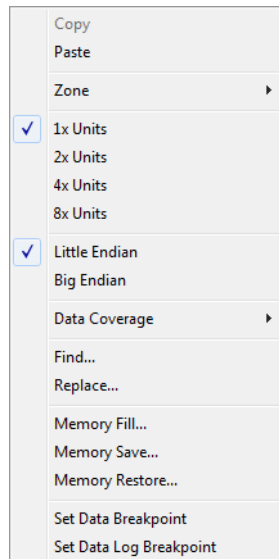
Data coverage is displayed with these colors:

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

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

Context menu

This context menu is available:



These commands are available:

Copy, Paste

Standard editing commands.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 134.

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 AVR*.

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 AVR*.

Memory Fill

Displays a dialog box, where you can fill a specified area with a value, see *Fill dialog box*, page 145.

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 143.

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 144.

Set Data Breakpoint

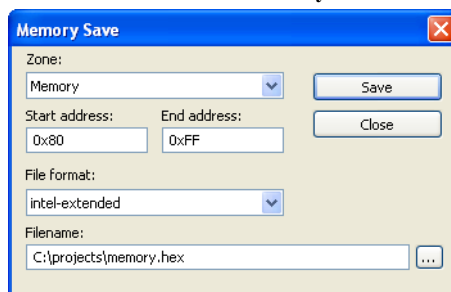
Sets breakpoints directly in the **Memory** window. The breakpoint is not highlighted; you can see, edit, and remove it in the **Breakpoints** dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the Memory window*, page 113.

Set Data Log Breakpoint

Sets a breakpoint on the start address of a memory selection directly in the **Memory** window. The breakpoint is not highlighted; you can see, edit, and remove it in the **Breakpoints** dialog box. The breakpoints you set in this window will be triggered by both read and write accesses; to change this, use the **Breakpoints** window. For more information, see *Data Log breakpoints*, page 107 and *Getting started using data logging*, page 179.

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

None; this dialog box is always available.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 134.

Start address

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

End address

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

File format

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

Filename

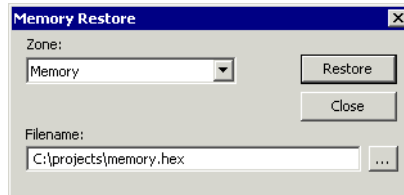
Specify the destination file to be used; a browse button is available for your convenience.

Save

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

Memory Restore dialog box

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



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

Requirements

None; this dialog box is always available.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 134.

Filename

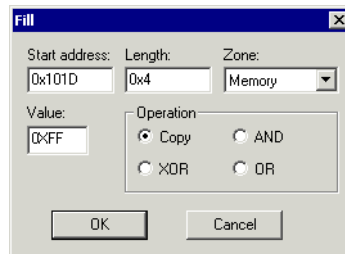
Specify the file to be read; a browse button is available for your convenience.

Restore

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

Fill dialog box

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



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

Requirements

None; this dialog box is always available.

Start address

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

Length

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

Zone

Selects a memory zone, see *C-SPY memory zones*, page 134.

Value

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

Operation

These are the available memory fill operations:

Copy

Value will be copied to the specified memory area.

AND

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

XOR

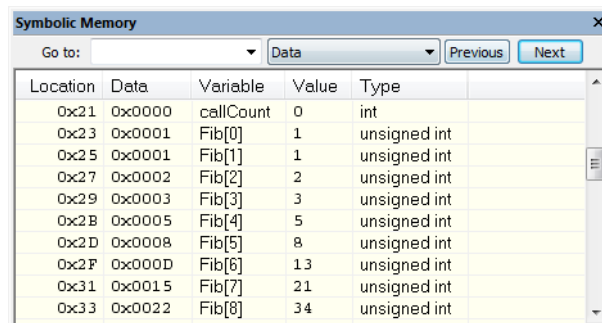
An **XOR** operation will be performed between Value and the existing contents of memory before writing the result to memory.

OR

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

Symbolic Memory window

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



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

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



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

See also *Editing in C-SPY windows*, page 54.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 134.

Previous

Highlights the previous symbol in the display area.

Next

Highlights the next symbol in the display area.

Display area

This area contains these columns:

Location

The memory address.

Data

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

Variable

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

Value

The value of the variable. This column is editable.

Type

The type of the variable.

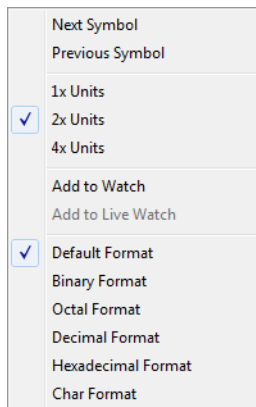
There are several different ways to navigate within the memory space:

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

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

Context menu

This context menu is available:



These commands are available:

Next Symbol

Highlights the next symbol in the display area.

Previous Symbol

Highlights the previous symbol in the display area.

1x Units

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

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

Add to Watch

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

Add to Live Watch

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

This command is not available in IAR Embedded Workbench for AVR.

Default format

Displays the memory contents in the default format.

Binary format

Displays the memory contents in binary format.

Octal format

Displays the memory contents in octal format.

Decimal format

Displays the memory contents in decimal format.

Hexadecimal format

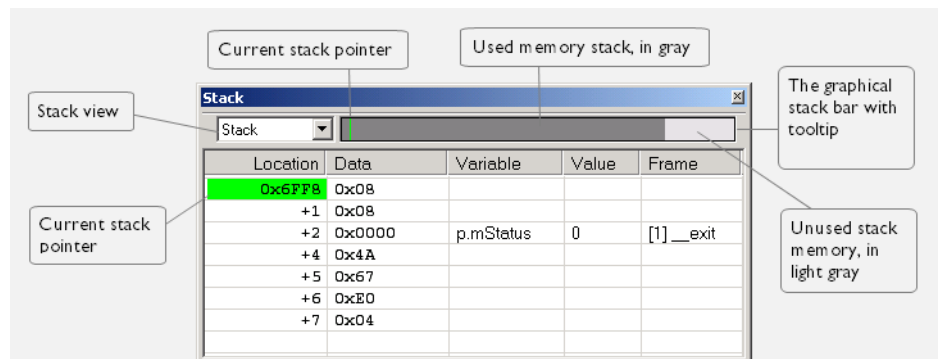
Displays the memory contents in hexadecimal format.

Char format

Displays the memory contents in char format.

Stack window

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



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

Note: By default, this window uses one physical breakpoint. For more information, see *Breakpoint consumers*, page 110.

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

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Stack

Selects which stack to view. This applies to microcontrollers with multiple stacks.

The graphical stack bar

Displays the state of the stack graphically.

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

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



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

Display area

This area contains these columns:

Location

Displays the location in memory. The addresses are displayed in increasing order. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.

Data

Displays the contents of the memory unit at the given location. From the **Stack** window context menu, you can select how the data should be displayed; as a 1-, 2-, or 4-byte group of data.

Variable

Displays the name of a variable, if there is a local variable at the given location. Variables are only displayed if they are declared locally in a function, and located on the stack and not in registers.

Value

Displays the value of the variable.

Type

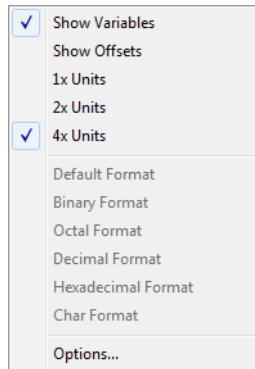
Displays the data type of the variable.

Frame

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

Context menu

This context menu is available:



These commands are available:

Show variables

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

Show offsets

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

1x Units

Displays the memory contents as single bytes.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

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

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

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

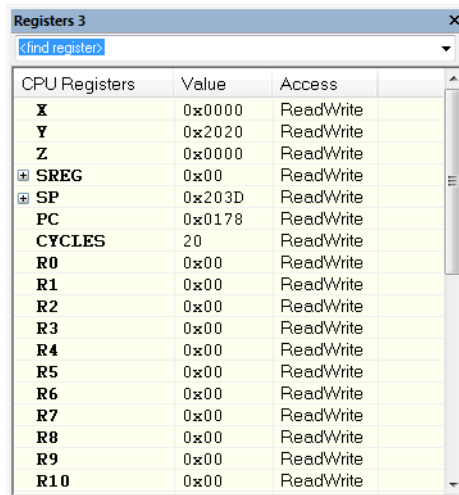
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration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

Options

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

Registers window

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



CPU Registers	Value	Access
X	0x0000	ReadWrite
Y	0x2020	ReadWrite
Z	0x0000	ReadWrite
+ SREG	0x00	ReadWrite
+ SP	0x203D	ReadWrite
PC	0x0178	ReadWrite
CYCLES	20	ReadWrite
R0	0x00	ReadWrite
R1	0x00	ReadWrite
R2	0x00	ReadWrite
R3	0x00	ReadWrite
R4	0x00	ReadWrite
R5	0x00	ReadWrite
R6	0x00	ReadWrite
R7	0x00	ReadWrite
R8	0x00	ReadWrite
R9	0x00	ReadWrite
R10	0x00	ReadWrite

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

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

See also *Editing in C-SPY windows*, page 54.

To enable predefined register groups:

- 1 Select a device description file that suits your device, see *Selecting a device description file*, page 51.
- 2 Display the register groups that are defined in the device description file in the **Registers** window 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 136.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

<find register>

Specify the name of a register that you want to find. Press the Enter key and the first register group where this register is found is displayed. User-defined register groups are not searched. The register search box has a history depth of 20 search entries.

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.

Register group 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 it 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

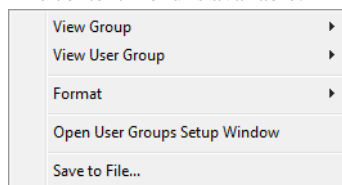
The access type of the register. Some of the registers are read-only, some of the registers are write-only.

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

CYCLES Cleared when an application is started or reset and is 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, by default **CPU Registers**. Additional predefined register groups are predefined in the device description files that make SFR registers available in the **Registers** windows. The device description file contains a section that defines the special function registers and their groups.

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 136.

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

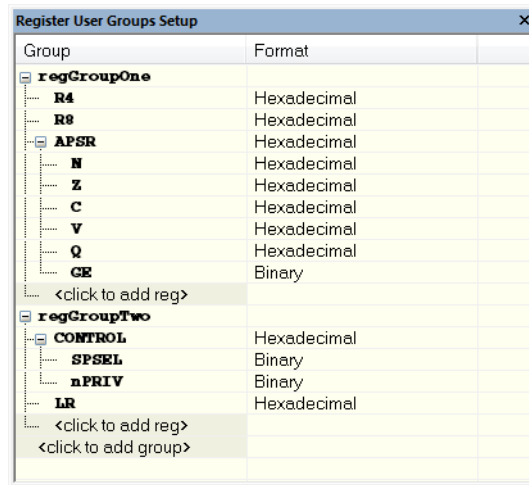
Opens a window where you can create your own user-defined register groups, see *Register User Groups Setup window*, page 156.

Save to File

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

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

None; this window is always available.

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.

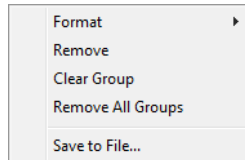
A dimmed register name indicates that it is not supported by the selected device.

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.

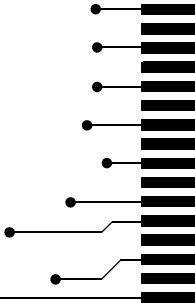
Save to File

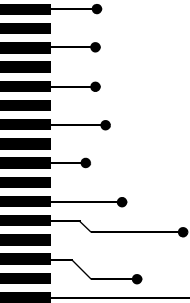
Opens a standard save dialog box to save the contents of the window to a tab-separated text file.

Part 2. Analyzing your application

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

- Trace
- The application timeline
- Profiling
- Code coverage
- Power debugging





Trace

- Introduction to using trace
- Collecting and using trace data
- Reference information on trace

Introduction to using trace

These topics are covered:

- Reasons for using trace
- Briefly about trace
- Requirements for using trace

See also:

- *Getting started using data logging*, page 179
- *Power debugging*, page 211
- *Profiling*, page 197

REASONS FOR USING TRACE

By using trace, you can inspect the program flow up to a specific state, for instance an application crash, and use the trace data to locate the origin of the problem. Trace data can be useful for locating programming errors that have irregular symptoms and occur sporadically.

BRIEFLY ABOUT TRACE

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

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

Trace features in C-SPY

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

Depending on your C-SPY driver, you:

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

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

REQUIREMENTS FOR USING TRACE

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

Trace data cannot be collected from the hardware debugger systems.

Collecting and using trace data

These tasks are covered:

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

GETTING STARTED WITH TRACE



- 1 After you have built your application and started C-SPY, open the **Trace** window—available from the driver-specific menu—and click the **Activate** button to enable collecting trace data.
- 2 Start the execution. When the execution stops, for example because a breakpoint is triggered, trace data is displayed in the **Trace** window. For more information about the window, see *Trace window*, page 164.

TRACE DATA COLLECTION USING BREAKPOINTS

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

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

For more information about these breakpoints, see *Trace Start breakpoints dialog box*, page 168 and *Trace Stop breakpoints dialog box*, page 169, respectively.

SEARCHING IN TRACE DATA

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

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

The **Find in Trace** window is very similar to the **Trace** window, showing the same columns and data, but *only* those rows that match the specified search criteria. Double-clicking an item in the **Find in Trace** window brings up the same item in the **Trace** window.

To search in your trace data:



- 1 On the **Trace** window toolbar, click the **Find** button.
- 2 In the **Find in Trace** dialog box, specify your search criteria.

Typically, you can choose to search for:

- A specific piece of text, for which you can apply further search criteria
- An address range
- A combination of these, like a specific piece of text within a specific address range.

For more information about the various options, see *Find in Trace dialog box*, page 172.

- 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 173.

BROWSING THROUGH TRACE DATA

To follow the execution history, simply look and scroll in the **Trace** window. Alternatively, you can enter *browse mode*.



- To enter browse mode, double-click an item in the **Trace** window, or click the **Browse** toolbar button.

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

Double-click again to leave browse mode.

Reference information on trace

Reference information about:

- *Trace window*, page 164
- *Function Trace window*, page 167
- *Trace Start breakpoints dialog box*, page 168
- *Trace Stop breakpoints dialog box*, page 169
- *Trace Expressions window*, page 170
- *Find in Trace dialog box*, page 172
- *Find in Trace window*, page 173.

Trace window

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

This window displays the collected trace data.

See also *Collecting and using trace data*, page 162.

Requirements

The C-SPY simulator.

Trace toolbar

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



Enable/Disable

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



Clear trace data

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



Toggle source

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



Browse

Toggles browse mode on or off for a selected item in the **Trace** window.

**Find**

Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 172.

**Save**

Displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.

**Edit Settings**

In the C-SPY simulator, this button is not enabled.

Edit Expressions (C-SPY simulator only)

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

**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

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

#	Cycles	Trace	callCount
1396	2766	FFF80574 MOV.L SP, R2	10
1397	2767	FFF80576 MOV #0x01:4, R1	10
1398	2771	FFF80578 BSR.A __DebugBreak	10
__DebugBreak:			
1399	2777	FFF804E0 RTS	10
1400	2780	FFF8057C BRA.B 0xFFF80572	10
1401	2781	FFF80572 MOV.L R6, [SP]	10
1402	2782	FFF80574 MOV.L SP, R2	10
1403	2783	FFF80576 MOV #0x01:4, R1	10

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

#

A serial number for each row in the trace buffer. Simplifies the navigation within the buffer.

Cycles

The number of cycles elapsed to this point.

Trace

The collected sequence of executed machine instructions. Optionally, the corresponding source code can also be displayed.

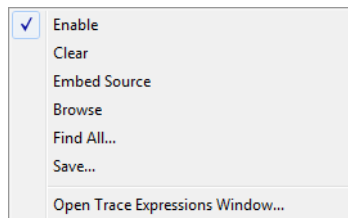
Expression

Each expression you have defined to be displayed appears in a separate column. Each entry in the expression column displays the value *after* executing the instruction on the same row. You specify the expressions for which you want to collect trace data in the **Trace Expressions** window, see *Trace Expressions window*, page 170.

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

Context menu

This context menu is available:



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

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.

Find All

Displays a dialog box where you can perform a search in the Trace window, see *Find in Trace dialog box*, page 172. The search results are displayed in the **Find in Trace** window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 173.

Save

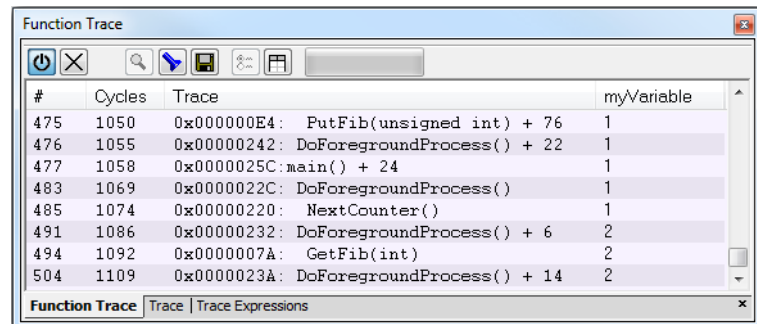
Displays a standard **Save As** dialog box where you can save the collected trace data to a text file, with tab-separated columns.

Open Trace Expressions Window

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

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

The C-SPY simulator.

Toolbar

For information about the toolbar, see *Trace window*, page 164.

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:

Cycles

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

Address

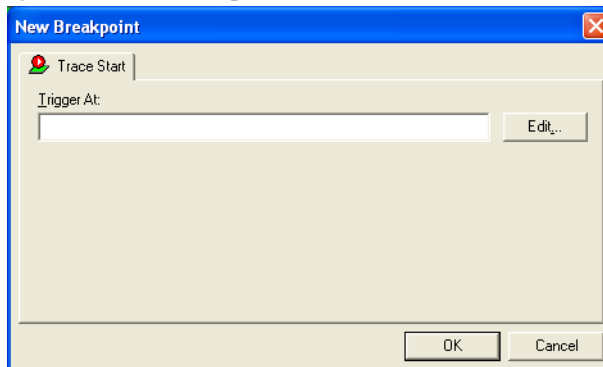
The address of the executed instruction.

Call/Return

The function that was called or returned to.

Trace Start breakpoints dialog box

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



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

See also *Trace Stop breakpoints dialog box*, page 169 and *Trace data collection using breakpoints*, page 162.

To set a Trace Start breakpoint:

- 1 In the editor or **Disassembly** window, right-click and choose **Trace Start** from the context menu.
Alternatively, open the **Breakpoints** window by choosing **View>Breakpoints**.
- 2 In the **Breakpoints** window, right-click and choose **New Breakpoint>Trace Start**.

Alternatively, to modify an existing breakpoint, select a breakpoint in the **Breakpoints** window and choose **Edit** on the context menu.

- 3 In the **Trigger At** text box, specify an expression, an absolute address, or a source location. Click **OK**.
- 4 When the breakpoint is triggered, the trace data collection starts.

Requirements

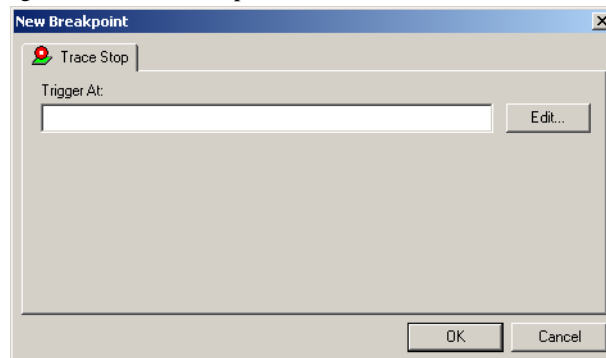
The C-SPY simulator.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 130.

Trace Stop breakpoints dialog box

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



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

See also *Trace Start breakpoints dialog box*, page 168 and *Trace data collection using breakpoints*, page 162.

To set a Trace Stop breakpoint:

- 1 In the editor or **Disassembly** window, right-click and choose **Trace Stop** from the context menu.

Alternatively, open the **Breakpoints** window by choosing **View>Breakpoints**.

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

Requirements

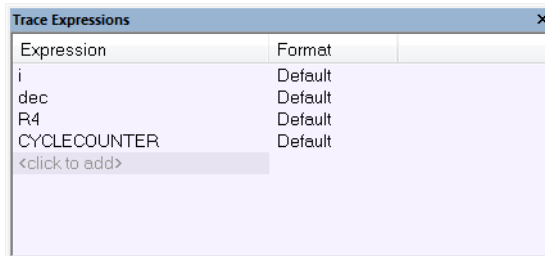
The C-SPY simulator.

Trigger at

Specify the code location of the breakpoint. Alternatively, click the **Edit** button to open the **Enter Location** dialog box, see *Enter Location dialog box*, page 130.

Trace Expressions window

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



Use this window to specify, for example, a specific variable (or an expression) for which you want to collect trace data.

Requirements

The C-SPY simulator.

Display area

Use the display area to specify expressions for which you want to collect trace data:

Expression

Specify any expression that you want to collect data from. You can specify any expression that can be evaluated, such as variables and registers.

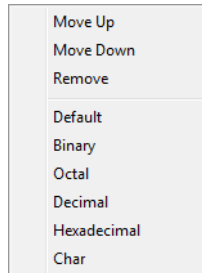
Format

Shows which display format that is used for each expression. Note that you can change display format via the context menu.

Each row in this area will appear as an extra column in the **Trace** window.

Context menu

This context menu is available:



These commands are available:

Move Up

Moves the selected expression upward in the window.

Move Down

Moves the selected expression downward in the window.

Remove

Removes the selected expression from 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:

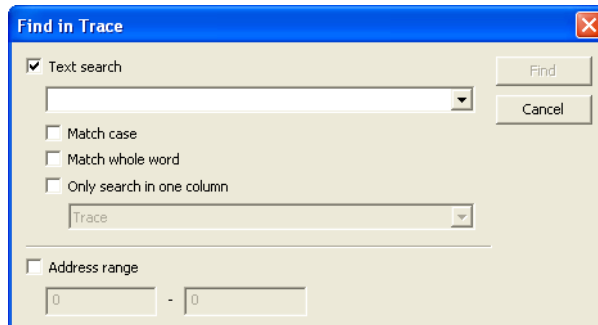
Variables	The display setting affects only the selected variable, not other variables.
-----------	--

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.

Find in Trace dialog box

The **Find in Trace** dialog box is available by clicking the **Find** button on the **Trace** window toolbar or by choosing **Edit>Find and Replace>Find**.

Note that the **Edit>Find and Replace>Find** command is context-dependent. It displays the **Find in Trace** dialog box if the **Trace** window is the current window or the **Find** dialog box if the editor window is the current window.



Use this dialog box to specify the search criteria for advanced searches in the trace data.

The search results are displayed in the **Find in Trace** window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 173.

See also *Searching in trace data*, page 163.

Requirements

The C-SPY simulator.

Text search

Specify the string you want to search for. To specify the search criteria, choose between:

Match Case

Searches only for occurrences that exactly match the case of the specified text. Otherwise **int** will also find **INT** and **Int** and so on.

Match whole word

Searches only for the string when it occurs as a separate word. Otherwise **int** will also find **print**, **sprintf** and so on.

Only search in one column

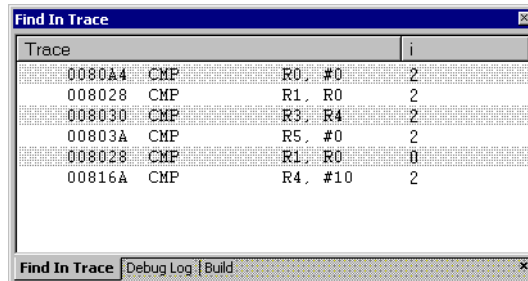
Searches only in the column you selected from the drop-down list.

Address Range

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

Find in Trace window

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



This window displays the result of searches in the trace data. Double-click an item in the **Find in Trace** window to bring up the same item in the **Trace** window.

Before you can view any trace data, you must specify the search criteria in the **Find in Trace** dialog box, see *Find in Trace dialog box*, page 172.

See also *Searching in trace data*, page 163.

Requirements

The C-SPY simulator.

Display area

The **Find in Trace** window looks like the **Trace** window and shows the same columns and data, but *only* those rows that match the specified search criteria.

The application timeline

- Introduction to analyzing your application's timeline
- Analyzing your application's timeline
- Reference information on application timeline

Introduction to analyzing your application's timeline

These topics are covered:

- Briefly about analyzing the timeline
- Requirements for timeline support

See also:

- *Trace*, page 161

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 75 and *Function Trace window*, page 167, 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.

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 211.

State logging Based on logged activity—state changes—for the GPIO input pins 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 211.

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	Power logging	State logging
C-SPY simulator	Yes	Yes	—	—
C-SPY Atmel-ICE driver	—	—	—	—
C-SPY AVR ONE! driver	—	—	—	—
C-SPY JTAGICE mkl/Dragon driver	—	—	—	—
C-SPY JTAGICE3 driver	—	—	—	—
C-SPY Power Debugger	—	—	Yes	Yes

Table 8: Support for timeline information

For more information about requirements related to trace data, see *Requirements for using trace*, page 162.

Analyzing your application's timeline

These tasks are covered:

- *Displaying a graph in the Timeline window*, page 177
- *Navigating in the graphs*, page 178
- *Analyzing performance using the graph data*, page 178
- *Getting started using data logging*, page 179

See also:

- *Debugging in the power domain*, page 217
- *Using the interrupt system*, page 246

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 175.

- 1 If you are using the Power Debugger, choose **Power Debugger>Power Debugger Settings** and select which of the four GPIO input pins that you want to monitor.
- 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 125.
- 6 Click **Go** on the toolbar to start executing your application. The graphs that you have enabled appear.

NAVIGATING IN THE GRAPHS

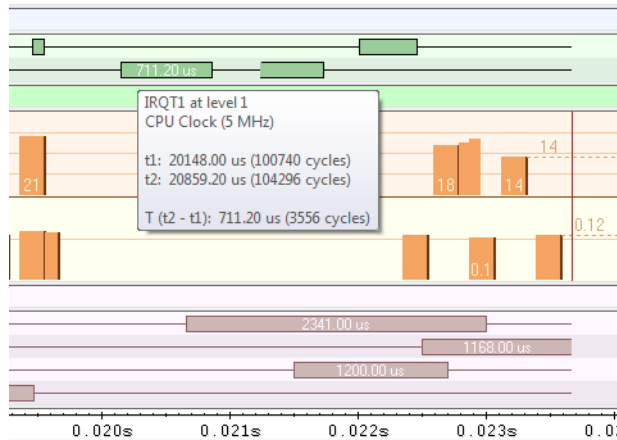
After you have performed the steps in *Displaying a graph in the Timeline window*, page 177, you can use any of these alternatives to navigate in the graph:

- 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

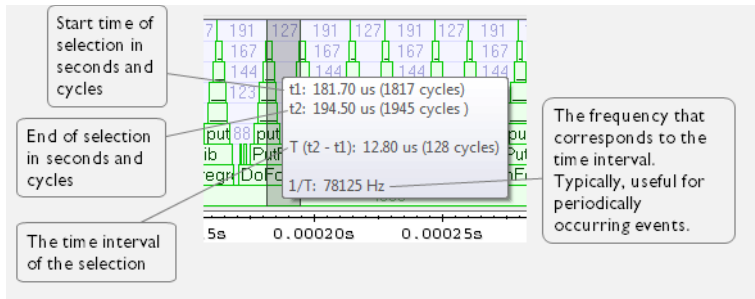
The **Timeline** window provides a set of tools for analyzing the graph data.

- 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 177.
- 3 Whenever execution stops, point at the graph with the mouse pointer to get detailed tooltip information for that location.



Note that if you have enabled several graphs, you can move the mouse pointer over the different graphs to get graph-specific information.

- 4 Click in the graph and drag to select a time interval. Point in the graph with the mouse pointer to get timing information for the selection.



GETTING STARTED USING DATA LOGGING

- I To set a data log breakpoint, use one of these methods:
 - In the **Breakpoints** window, right-click and choose **New Breakpoint>Data Log** to open the breakpoints dialog box. Set a breakpoint on the memory location that you want to collect log information for. This can be specified either as a variable or as an address.

- In the **Memory** window, select a memory area, right-click and choose **Set Data Log Breakpoint** from the context menu. A breakpoint is set on the start address of the selection.
- In the editor window, select a variable, right-click and choose **Set Data Log Breakpoint** from the context menu. The breakpoint will be set on the part of the variable that the microcontroller can access using one instruction.

You can set up to four data log breakpoints. For more information about data log breakpoints, see *Data Log breakpoints*, page 107.

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

Reference information on application timeline

Reference information about:

- *Timeline window—Call Stack graph*, page 181
- *Timeline window—Data Log graph*, page 184
- *Data Log window*, page 188
- *Data Log Summary window*, page 191
- *Viewing Range dialog box*, page 194

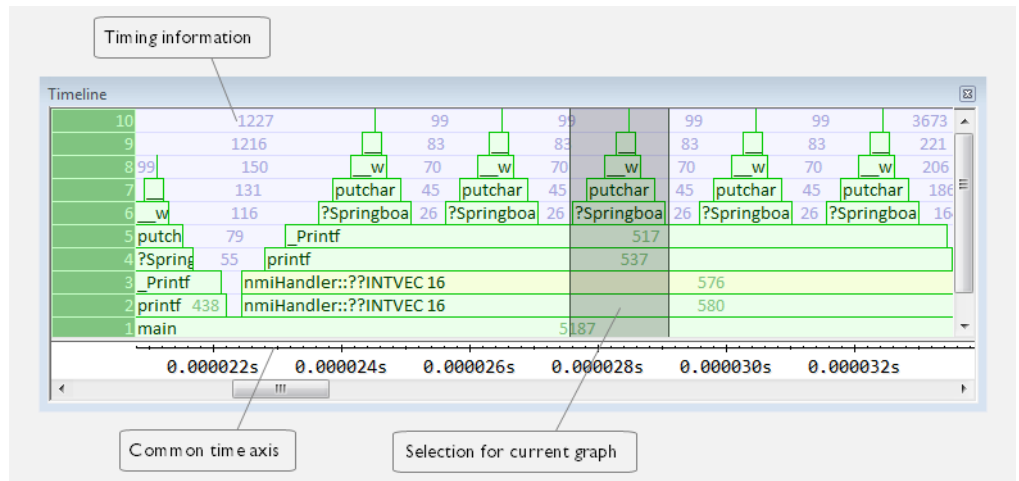
See also:

- *Timeline window—Power graph*, page 223
- *Power Log window*, page 226
- *Timeline window—State Log graph*, page 235

- *State Log window*, page 231
- *State Log Summary window*, page 233

Timeline window—Call Stack graph

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



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

The Call Stack graph displays the sequence of function calls and returns collected by the trace system.

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

Requirements

The C-SPY simulator.

Display area for the Call Stack graph

Each function invocation is displayed as a horizontal bar which extends from the time of entry until the return. Called functions are displayed above its caller. The horizontal bars use four different colors:

- Medium green for normal C functions with debug information

- Light green for functions known to the debugger only through an assembler label
- Medium yellow for normal interrupt handlers, with debug information
- Light yellow for interrupt handlers known to the debugger only through an assembler label

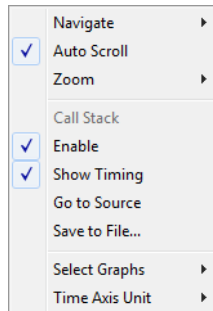
The timing information represents the number of cycles spent in, or between, the function invocations.

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.

Context menu

This context menu is available:



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

These commands are available:

Navigate

Commands for navigating the graph(s). Choose between:

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

Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll

Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.

Zoom

Commands for zooming the window, in other words, changing the time scale. Choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +

Zoom Out zooms out on the time scale. Shortcut key: –

10ns, 100ns, 1us, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

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

10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Call Stack

A heading that shows that the Call stack-specific commands below are available.

Enable

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.

Show Timing

Toggles the display of the timing information on or off.

Go To Source

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

Save to File

Saves all contents (or the selected contents) of the Call Stack graph to a file. The menu command is only available when C-SPY is not running.

Select Graphs

Selects which graphs to be displayed in the **Timeline** window.

Time Axis Unit

Selects the unit used in the time axis; choose between **Seconds** and **Cycles**.

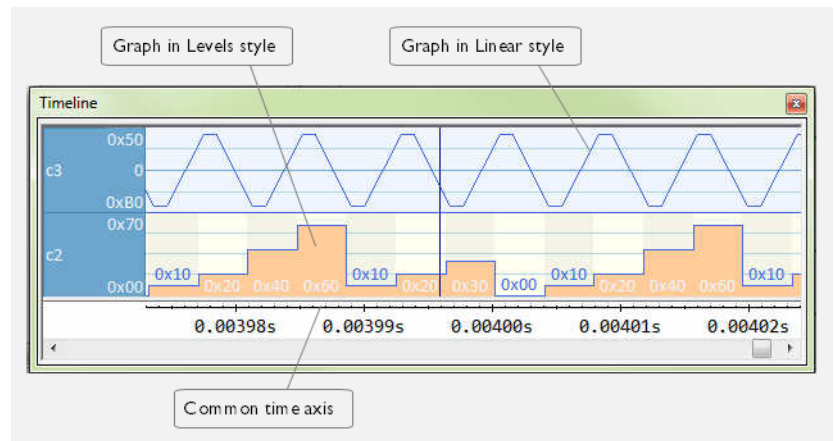
If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.

Profile Selection

Enables profiling time intervals in the **Function Profiler** window. Note that this command is only available if the C-SPY driver supports PC Sampling.

Timeline window—Data Log graph

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



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

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

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

Requirements

The C-SPY simulator.

Display area for the Data Log graph

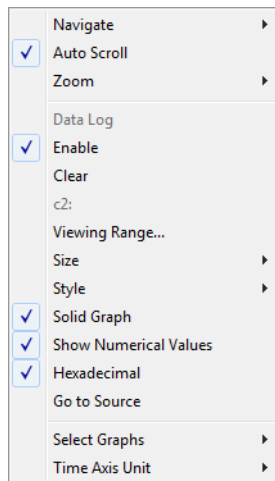
Where:

- The label area at the left end of the graph displays the variable name or the address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the Y-axis for a variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the **Data Log** window, see *Data Log window*, page 188.
- The graph can be displayed either as a thin line between consecutive logs or as a rectangle for every log (optionally color-filled).
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system. A red question mark indicates a log without a value.

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

Context menu

This context menu is available:



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

These commands are available:

Navigate

Commands for navigating the graph(s). Choose between:

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

Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll

Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.

Zoom

Commands for zooming the window, in other words, changing the time scale. Choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +

Zoom Out zooms out on the time scale. Shortcut key: –

10ns, 100ns, 1us, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

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

10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

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 happen also 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 194.

Size

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

Solid Graph

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

Show Numerical Value

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

Hexadecimal

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

Go To Source

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

Select Graphs

Selects which graphs to be displayed in the **Timeline** window.

Time Axis Unit

Selects the unit used in the time axis; choose between **Seconds** and **Cycles**.

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

Data Log window

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

Time	Program Counter	I1	Address	s2	Address
0.160us	---			W 0x0000	@ 0x2004
0.160us	0xFFE00049	-	@ 0x2000		
24.480us	0xFFE000B5			R 0x0000	@ 0x2006
24.720us	0xFFE000BF			W 0x0042	@ 0x2004
24.760us	0xFFE000C6			R 0x0042	@ 0x2006
24.960us	0xFFE000E4	W 0x00004444	@ 0x2000		
<i>78.760us</i>	0xFFE00104			R 0x0042	@ 0x2004+?
79.000us	---			W 0x0084	@ 0x2004
100.800us	0xFFE00104			R 0x0084	@ 0x2006
101.040us	0xFFE0010E			W 0x00C6	@ 0x2004
<i>136.640us</i>	Overflow				
136.880us	0xFFE0010E			-	@ 0x2004

White rows indicate read accesses

Grey rows indicate write accesses

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

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

See also *Getting started using data logging*, page 179.

Requirements

The C-SPY simulator.

Display area

Each row in the display area shows the time, the program counter, and, for every tracked data object, its value and address. All information is cleared on reset. The information is displayed in these columns:

Time

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

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

Cycles

The number of cycles from the start of the execution until the event.

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

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

Program Counter*

Displays one of these:

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

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

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

Value

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

To specify what data you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 107.

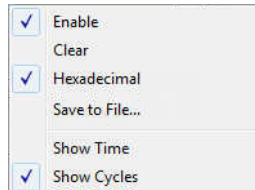
Address

The actual memory address that is accessed. For example, if only a byte of a word is accessed, only the address of the byte is displayed. The address is calculated as base address + offset, where the base address is retrieved from the **Data Log** breakpoint dialog box and the offset is retrieved from the logs. If the log from the target system does not provide the debugger with an offset, the offset contains + ?.

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

Context menu

This context menu is available:



These commands are available:

Enable

Enables the logging system. The system will log information also when the window is closed.

Clear

Deletes the log information. Note that this will happen also when you reset the debugger.

Hexadecimal

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

Save to File

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

Show Time

Displays the **Time** column.

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

Show Cycles

Displays the **Cycles** column.

If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Data Log Summary window

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

Data	Total Accesses	Read Accesses	Write Accesses	Unknown Accesses
tVar1	42	0	25	17
tVar2	66	17	49	0
tVar3	32	32	0	0

Approximative time count: 16
 Overflow count: 8
 Current time: 4301.52 us

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

See also *Getting started using data logging*, page 179.

Requirements

The C-SPY simulator.

Display area

Each row in this area displays the type and the number of accesses to each memory location or area in these columns. Summary information is listed at the bottom of the display area.

Data

The name of the data object you have selected to log accesses to. To specify what data object you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 107.

Total Accesses

The total number of accesses.

If the sum of read accesses and write accesses is less than the total accesses, the target system for some reason did not provide valid access type information for all accesses.

Read Accesses

The total number of read accesses.

Write Accesses

The total number of write accesses.

Unknown Accesses

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

Approximative time count

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

For some C-SPY drivers, this information is not displayed or the value is always zero. In this case, all logs have an exact time stamp.

For other C-SPY drivers, a non-zero value is displayed. The value represents the amount of logs with an approximative time stamp. This might happen if the bandwidth in the communication channel is too low compared to the amount of data packets generated by the CPU or if the CPU generated packets with an approximative time stamp.

Overflow count

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

For some C-SPY drivers, this information is not displayed or the value is always zero.

For other C-SPY drivers, the number represents the amount of overflows in the communication channel which can cause logs to be lost. If this happens, it indicates that logs might be incomplete. To solve this, make sure not to use all C-SPY log features simultaneously or check used bandwidth for the communication channel.

Current time|cycles

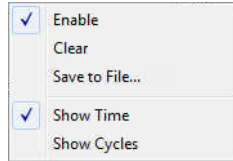
The information displayed depends on the C-SPY driver you are using.

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 happen also 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.

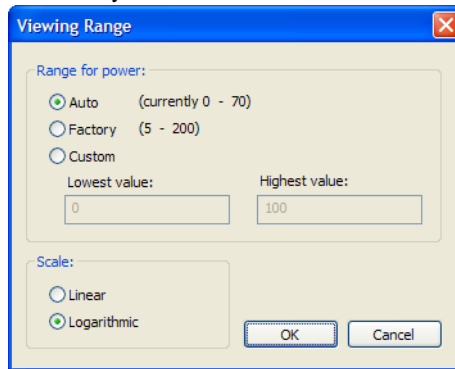
Show Cycles

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.

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.

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 the other graphs: Uses the range according to the value range of the variable, for example 0–65535 for an unsigned 16-bit integer.

Custom

Use the text boxes to specify an explicit range.

Scale

Selects the scale type of the Y-axis:

- **Linear**

- **Logarithmic.**

Profiling

- Introduction to the profiler
- Using the profiler
- Reference information on the profiler

Introduction to the profiler

These topics are covered:

- Reasons for using the profiler
- Briefly about the profiler
- Requirements for using the profiler

REASONS FOR USING THE PROFILER

Function profiling can help you find the functions in your source code where the most time is spent during execution. You should focus on those functions when optimizing your code. A simple method of optimizing a function is to compile it using speed optimization. Alternatively, you can move the data used by the function into more efficient memory. For detailed information about efficient memory usage, see the *IAR C/C++ Compiler Reference Guide for AVR*.

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

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

BRIEFLY ABOUT THE PROFILER

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

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

Profiling sources

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

- Trace (calls)

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

- Trace (flat)

Each instruction in the full instruction trace or each PC Sample is assigned to a corresponding function or code fragment, without regard to function calls or returns. This is most useful when the application does not exhibit normal call/return sequences, such as when you are using an RTOS, or when you are profiling code which does not have full debug information.

Power sampling

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

REQUIREMENTS FOR USING THE PROFILER

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

Using the profiler

These tasks are covered:

- Getting started using the profiler on function level
- Analyzing the profiling data

- Getting started using the profiler on instruction level

GETTING STARTED USING THE PROFILER ON FUNCTION LEVEL

To display function profiling information in the **Function Profiler** window:

- I Build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Format>Debug information for C-SPY

Table 9: Project options for enabling the profiler



- 2 When you have built your application and started C-SPY, choose **C-SPY driver>Function Profiler** to open the **Function Profiler** window, and click the **Enable** button to turn on the profiler. Alternatively, choose **Enable** from the context menu that is available when you right-click in the **Function Profiler** window.

- 3 Start executing your application to collect the profiling information.

- 4 Profiling information is displayed in the **Function Profiler** window. To sort, click on the relevant column header.



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

ANALYZING THE PROFILING DATA

Here follow some examples of how to analyze the data.

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

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

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

The screenshot shows the 'Function Profiler' window with a table of function calls. A context menu is open over the 'GetFib' function, showing options like 'Enable', 'Clear', 'Source: Trace (calls)', and 'Source: Trace (flat)'. Red circles highlight the '231' value in the 'Flat Time' column and the '487' value in the 'Acc. Time' column for the 'GetFib' function.

Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
main	1	165	3.58	4356	94.39
DoForegroundProcess	10			3704	
InitFib	1			487	
PutFib	10	3174	68.78	3174	68.78
NextCounter	10	100	2.17	100	2.17
InitFib	1	231	5.01	487	10.55
GetFib	16			256	
GetFib	26	416	9.01	416	9.01
DoForegroundProcess	10	270	5.85	3704	80.26
GetFib	10			160	
NextCounter	10				
PutFib	10				
<Other>	0	25			98.85
main	1				

The second figure shows the result of profiling using **Source: Trace (flat)**. In this case, the profiler does not follow the program flow, instead the profiler only detects whether the PC address is within the function scope. For incomplete trace data, the data might contain minor errors.

For the **InitFib** function, **Flat Time** 231 is the time (number of hits) spent inside the function itself.

Function	PC Samp...	PC Samples ...
<Idle>	0	0.00
<No function>	5	0.21
DoForegroundProcess	90	3.85
GetFib	260	11.12
InitFib	141	6.03
NextCounter	60	2.57
PutFib	230	9.84
__cmain, ?main	4	0.17
__default_handler, NML_H...	0	0.00
__dwrite		
__exit		
__jar_close_ttio		
__jar_copy_init3		
__jar_data_init3		
__jar_get_ttio		
__jar_lookup_ttioh		
__jar_sh_stdout		

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

GETTING STARTED USING THE PROFILER ON INSTRUCTION LEVEL

To display instruction profiling information in the Disassembly window:

- 1 When you have built your application and started C-SPY, choose **View>Disassembly** to open the **Disassembly** window, and choose **Instruction Profiling>Enable** from the context menu that is available when you right-click in the left-hand margin of the **Disassembly** window.
- 2 Make sure that the **Show** command on the context menu is selected, to display the profiling information.
- 3 Start executing your application to collect the profiling information.
- 4 When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, you can view instruction level profiling information in the left-hand margin of the window.

```

Disassembly
Go to: main Memory
Dly100us:
text_12:
0 08005F92 B082 SUB SP, SP, #0x8
Int32U_Dly = (Int32U)arg;
0 08005F94 E005 B ??Dly100us_0
for(volatile int i = LOOP_DLY_100US; i; i--):
??Dly100us_1:
34 08005F96 9900 LDR R1, [SP]
5 08005F98 1E49 SUBS R1, R1, #0x1
5 08005F9A 9100 STR R1, [SP]
for(volatile int i = LOOP_DLY_100US; i; i--):
??Dly100us_2:
11 08005F9C 9900 LDR R1, [SP]
3 08005F9E 2900 CMP R1, #0x0
34 08005FA0 D1F9 BNE ??Dly100us_1
while(Dly--)
??Dly100us_0:
0 08005FA2 0001 MOVS R1, R0
0 08005FA4 1E48 SUBS R0, R1, #0x1

```

For each instruction, the number of times it has been executed is displayed.

Reference information on the profiler

Reference information about:

- *Function Profiler window*, page 202

See also:

- *Disassembly window*, page 71

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 (%)
main()	1	165	3.57	4356	94.18
PutFib(unsigned int)	10	3174	68.63	3174	68.63
NextCounter()	10	100	2.16	100	2.16
InitFib()	1	231	4.99	487	10.53
GetFib(int)	26	416	8.99	416	8.99
DoForegroundProcess()	10	270	5.84	3704	80.09
<Other>	0	269	5.82	4572	98.85

This window displays function profiling information.

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

See also *Using the profiler*, page 198.

Requirements

The C-SPY simulator.

Toolbar

The toolbar contains:



Enable/Disable

Enables or disables the profiler.



Clear

Clears all profiling data.



Save

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



Graphical view

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

Progress bar

Displays a backlog of profiling data that is still being processed. If the rate of incoming data is higher than the rate of the profiler processing the data, a backlog is accumulated. The progress bar indicates that the profiler is still processing data, but also approximately how far the profiler has come in the process. Note that because the profiler consumes data at a certain rate and the target system supplies data at another rate, the amount of data remaining to be processed can both increase and decrease. The progress bar can grow and shrink accordingly.

Display area

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

- For the Trace (calls) source, the display area contains one line for each function compiled with debug information enabled. When some profiling information has been collected, it is possible to expand rows of functions that have called other

functions. The child items for a given function list all the functions that have been called by the parent function and the corresponding statistics.

- For the Trace (flat) source, the display area contains one line for each C function of your application, but also lines for sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels. Each executed PC address from trace data is treated as a separate sample and is associated with the corresponding line in the Profiling window. Each line contains a count of those samples.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 198.

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

Function (All sources)

The name of the profiled C function.

Calls (Trace (calls))

The number of times the function has been called.

Flat time (Trace (calls))

The time expressed as the estimated number of 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 as the estimated number of 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 Samples (Trace (flat))

The number of PC samples associated with the function.

PC Samples (%) (Trace (flat))

The number of PC samples associated with the function as a percentage of the total number of samples.

Power Samples (Power Sampling)

The number of power samples associated with that function.

Energy (%) (Power Sampling)

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

Avg Current [mA] (Power Sampling)

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

Min Current [mA] (Power Sampling)

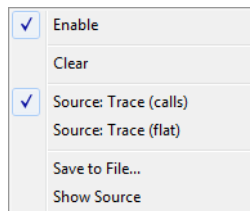
The minimum measured value for all samples associated with that function.

Max Current [mA] (Power Sampling)

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

Context menu

This context menu is available:



The contents of this menu depend on the C-SPY driver you are using.

These commands are available:

Enable

Enables the profiler. The system will collect information also when the window is closed.

Clear

Clears all profiling data.

Filtering

Selects which part of your code to profile. Choose between:

Check All—Excludes all lines from the profiling.

Uncheck All—Includes all lines in the profiling.

Load—Reads all excluded lines from a saved file.

Save—Saves all excluded lines to a file. Typically, this can be useful if you are a group of engineers and want to share sets of exclusions.

These commands are only available when using Trace (flat).

Source*

Selects which source to be used for the profiling information. See also *Profiling sources*, page 198. Choose between:

Trace (calls)—the instruction count for instruction profiling is only as complete as the collected trace data.

Trace (flat)—the instruction count for instruction profiling is only as complete as the collected trace data.

Power Sampling

Toggles power sampling information on or off.

Save to File

Saves all profiling data to a file.

Show Source

Opens the editor window (if not already opened) and highlights the selected source line.

* The available sources depend on the C-SPY driver you are using.

Code coverage

- Introduction to code coverage
- Reference information on code coverage.

Introduction to code coverage

These topics are covered:

- Reasons for using code coverage
- Briefly about code coverage
- Requirements and restrictions for using code coverage.

REASONS FOR USING CODE COVERAGE

The code coverage functionality is useful when you design your test procedure to verify whether all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

BRIEFLY ABOUT CODE COVERAGE

The **Code Coverage** window reports the status of the current code coverage analysis for C code. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

Note: Assembler code is not covered by the code coverage analysis. To view assembler code, use the **Disassembly** window.

REQUIREMENTS AND RESTRICTIONS FOR USING CODE COVERAGE

Code coverage is supported by the C-SPY Simulator and there are no specific requirements or restrictions.

Reference information on code coverage

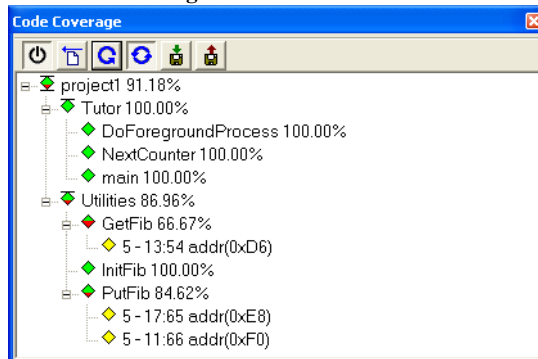
Reference information about:

- *Code Coverage window*, page 208.

See also *Single stepping*, page 64.

Code Coverage window

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



This window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

An asterisk (*) in the title bar indicates that C-SPY has continued to execute, and that the **Code Coverage** window must be refreshed because the displayed information is no longer up to date. To update the information, use the **Refresh** button.

To get started using code coverage:

- 1 Before using the code coverage functionality you must build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Format>Debug information for C-SPY
Debugger	Plugins>Code Coverage

Table 10: Project options for enabling code coverage

- 2 After you have built your application and started C-SPY, choose **View>Code Coverage** to open the **Code Coverage** window.
- 3 Click the **Activate** button, alternatively choose **Activate** from the context menu, to switch on code coverage.





- 4 Start the execution. When the execution stops, for instance because the program exit is reached or a breakpoint is triggered, click the **Refresh** button to view the code coverage information.

Requirements

The C-SPY simulator.

Display area

The code coverage information is displayed in a tree structure, showing the program, module, function, and statement levels. The window displays only source code that was compiled with debug information. Thus, startup code, exit code, and library code is not displayed in the window. Furthermore, coverage information for statements in inlined functions is not displayed. Only the statement containing the inlined function call is marked as executed. The plus sign and minus sign icons allow you to expand and collapse the structure.

These icons give you an overview of the current status on all levels:

Red diamond	Signifies that 0% of the modules or functions has been executed.
Green diamond	Signifies that 100% of the modules or functions has been executed.
Red and green diamond	Signifies that some of the modules or functions have been executed.
Yellow diamond	Signifies a statement that has not been executed.

The percentage displayed at the end of every program, module, and function line shows the amount of statements that has been covered so far, that is, the number of executed statements divided with the total number of statements.

For statements that have not been executed (yellow diamond), the information displayed is the column number range and the row number of the statement in the source window, followed by the address of the step point:

```
<column_start>-<column_end>: row address.
```

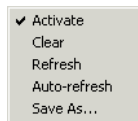
A statement is considered to be executed when one of its instructions has been executed. When a statement has been executed, it is removed from the window and the percentage is increased correspondingly.

Double-clicking a statement or a function in the **Code Coverage** window displays that statement or function as the current position in the editor window, which becomes the

active window. Double-clicking a module on the program level expands or collapses the tree structure.

Context menu

This context menu is available:



These commands are available:



Activate

Switches code coverage on and off during execution.



Clear

Clears the code coverage information. All step points are marked as not executed.



Refresh

Updates the code coverage information and refreshes the window. All step points that have been executed since the last refresh are removed from the tree.



Auto-refresh

Toggles the automatic reload of code coverage information on and off. When turned on, the code coverage information is reloaded automatically when C-SPY stops at a breakpoint, at a step point, and at program exit.

Save As

Saves the current code coverage result in a text file.



Save session

Saves your code coverage session data to a *.dat file. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command might not be supported by the C-SPY driver you are using.



Restore session

Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command might not be supported by the C-SPY driver you are using.

Power debugging

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

Introduction to power debugging

These topics are covered:

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

REASONS FOR USING POWER DEBUGGING

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

For examples of when power debugging can be useful, see *Optimizing your source code for power consumption*, page 213.

BRIEFLY ABOUT POWER DEBUGGING

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

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

Measuring power consumption

Power debugging using C-SPY

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

- The **Power Setup** window is where you can specify a threshold and an action to be executed when the threshold is reached. This means that you can enable or disable the power measurement or you can stop the application's execution and determine the cause of unexpected power values.
- The **Power Log** window displays all logged power values. This window can be used for finding peaks in the power logging.
- 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 **State Log** window logs activity—state changes—for peripheral units and clocks, as well as for CPU modes. The **State Log Summary** window displays a summary of the logged activity. The State Log graphs display a graphical view of the activity. The information is useful for tracing the activity on the target system.

Power debugging with the Atmel Power Debugger

The Atmel Power Debugger has a designated circuit for sampling electrical current, electrical voltage, and GPIO input levels. IAR Embedded Workbench for AVR can display this data as a graph in the **Timeline** window and as a list of sampled values in the **Power Log** window and the **State Log** window.

Because all samples are timestamped using the same clock on the power debugger probe, the time accuracy is very high*. There is no direct correlation between the samples and the program counter, but you can manually instrument your code by setting and clearing an output pin on the chip connected to a GPIO pin on the debug probe, to see how a certain code section in your application affects the power consumption.

* Note that the time accuracy is impaired when the execution starts after a halt, because then the starting time needs to be approximated. This means that you cannot expect the same time accuracy when comparing a timestamp of a sample from within one execution period (from a **Go** command to the next stop) with a timestamp from another execution period. However, the accuracy of timestamps within an execution period is very high.

REQUIREMENTS AND RESTRICTIONS FOR POWER DEBUGGING

To use the features in C-SPY for power debugging, you need the Atmel Power Debugger.

Optimizing your source code for power consumption

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

These topics are covered:

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

WAITING FOR DEVICE STATUS

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

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

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

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

SOFTWARE DELAYS

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

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

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

LOW-POWER MODE DIAGNOSTICS

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

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

CPU FREQUENCY

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

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

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

Power debugging lets you verify the power consumption as a factor of the clock frequency. A system that spends very little time in sleep mode at MHz is expected to spend 50% of the time in sleep mode when running at 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. In many cases, it is enough to disable the peripheral unit when it is inactive, for example by turning off its clock which in most cases will shut down its power consumption completely.

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

PERIPHERAL UNITS IN AN EVENT-DRIVEN SYSTEM

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

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



At t_1 , the system is activated whereby the current rises to I_1 which is the system's power consumption in active mode when at least one peripheral device turned on, causing the

current to rise to I_1 . At t_2 , the execution becomes suspended by an interrupt which is handled with high priority. Peripheral devices that were already active are not turned off, although the task with higher priority is not using them. Instead, more peripheral devices are activated by the new task, resulting in an increased current I_2 between t_2 and t_3 where control is handed back to the task with lower priority.

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

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

FINDING CONFLICTING HARDWARE SETUPS

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

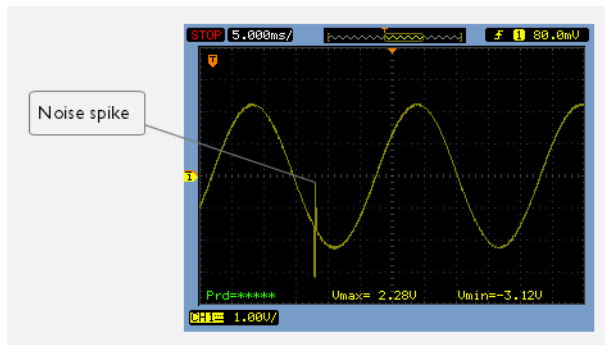
A similar situation arises if an I/O pin is designed to be an input and is driven by an external circuit, but your code incorrectly configures the input pin as output.

ANALOG INTERFERENCE

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

requires careful hardware considerations. Your software design can also affect the quality of the analog measurements.

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



Power debugging will help you investigate interference from digital and power supply lines into the analog parts. Power spikes in the vicinity of AD conversions could be the source of noise and should be investigated.

Debugging in the power domain

These tasks are covered:

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

See also:

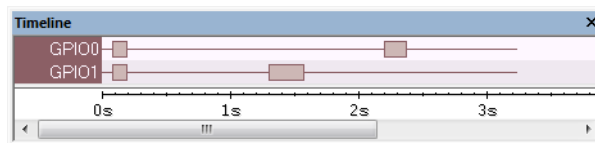
- *Timeline window—Power graph*, page 223

DISPLAYING A POWER PROFILE AND ANALYZING THE RESULT

To view the power profile:

- 1 Choose **Atmel Power Debugger>Power Debugging Settings** to open the **Power Debugging Settings** dialog box. Use the dialog box to select which of the four GPIO input pins on the Atmel Power Debugger probe to monitor.
- 2 Start the debugger.

- 3 Choose **C-SPY driver>Power Log Setup**. In the **Wanted** text box, you can set an upper limit for the sampling frequency. In the **ID** column, make sure to select the alternatives for which you want to enable power logging.
- 4 Choose **C-SPY driver>Timeline** to open the **Timeline** window.
- 5 Right-click in the graph area and choose **Enable** from the context menu to enable the power graph you want to view.
- 6 Choose **C-SPY driver>Power Log** to open the **Power Log** window.
- 7 Optionally, if you want to correlate power values to the status of the four GPIO input pins, right-click in the State Log graph area, and choose **Enable** from the context menu.
- 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 194.
- 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 a graphical representation of the logged activity—state changes—for the four GPIO input pins if you enabled the State Log graph. For information about how to navigate on the graph, see *Navigating in the graphs*, page 178.



10 To analyze power consumption:

- 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 215.
- 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.

Note: To analyze power consumption as described above, you need to instrument your code. For more information, see *Power debugging with the Atmel Power Debugger*, page 212.

DETECTING UNEXPECTED POWER USAGE DURING APPLICATION EXECUTION

To detect unexpected power consumption:

- 1 Choose **Atmel Power Debugger>Power Debugging Settings** to open the **Power Debugging Settings** dialog box. Use the dialog box to select which GPIO input pins to monitor.
- 2 Choose **C-SPY driver>Power Log Setup** to open the **Power Setup** window.
- 3 In the **Power Setup** window, specify a threshold value and the appropriate action, for example **Log All** and **Halt CPU Above Threshold**.
- 4 Choose **C-SPY driver>Power Log** to open the **Power Log** window. If you continuously want to save the power values to a file, choose **Choose Live Log File** from the context menu. In this case you also need to choose **Enable Live Logging to**.
- 5 Start the execution.

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

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

CHANGING THE GRAPH RESOLUTION

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

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

Reference information on power debugging

Reference information about:

- *Power Log Setup window*, page 220

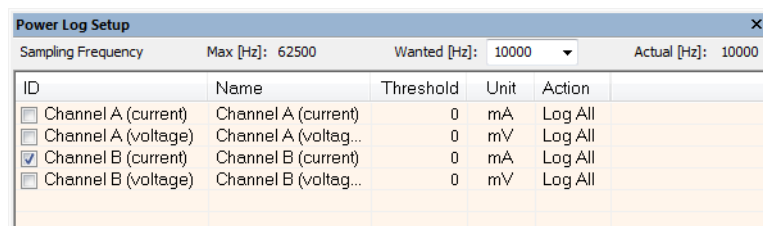
- *Power Debugging Settings*, page 222
- *Power Log window*, page 226.
- *Timeline window—Power graph*, page 223.
- *State Log Setup window*, page 229.
- *State Log window*, page 231.
- *State Log Summary window*, page 233.
- *Timeline window—State Log graph*, page 235.

See also:

- *Trace window*, page 164
- *The application timeline*, page 175
- *Viewing Range dialog box*, page 194
- *Function Profiler window*, page 202.

Power Log Setup window

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



Use this window to configure the power measurement.

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

See also *Debugging in the power domain*, page 217.

Requirements

The C-SPY Power Debugger driver and the Power Debugger probe.

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.

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.

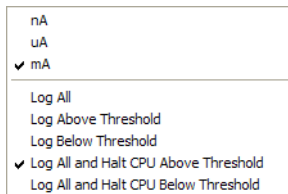
Action

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:

uA, mA

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

Log All

Logs all values.

Log Above Threshold

Logs all values above the threshold.

Log Below Threshold

Logs all values below the threshold.

Log All and Halt CPU Above Threshold

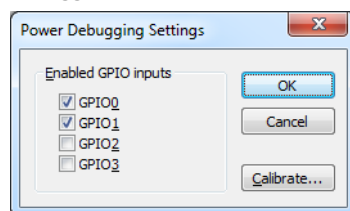
Logs all values. If a logged value exceeds the threshold, execution is stopped.

Log All and Halt CPU Below Threshold

Logs all values. If a logged value goes below the threshold, execution is stopped.

Power Debugging Settings

The **Power Debugging Settings** dialog box is available from the **Atmel Power Debugger** menu.



Requirements

The C-SPY Power Debugger driver and the Power Debugger probe.

Enabled GPIO inputs

Select which GPIO input pins to monitor.

Already logged GPIO states will be visible in the **Power Log** window until you clear the log manually or until the internal (circular) buffer is full.

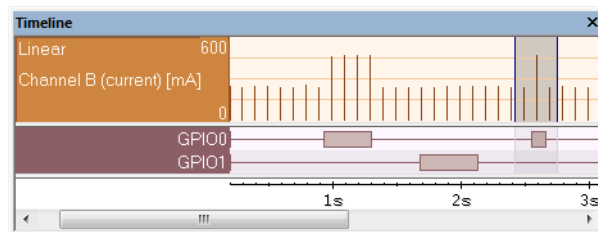
Calibrate

Calibrates the power measurement interface. This is recommended to improve the accuracy of the power measurements.

This option is only available in an active debug session.

Timeline window—Power graph

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



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

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

For more information about the **Timeline** window, how to display a graph, and the other supported graphs, see *The application timeline*, page 175.

See also *Requirements and restrictions for power debugging*, page 213.

Requirements

The C-SPY Power Debugger driver and the Power Debugger probe.

Display area

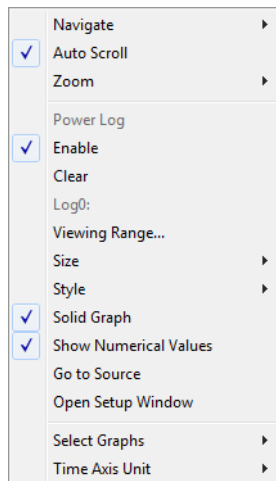
Where:

- The label area at the left end of the graph displays the name of the measurement channel.
- 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.

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

Context menu

This context menu is available:



Note: The exact contents of the context menu you see on the screen depends on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Navigate

Commands for navigating the graph(s). Choose between:

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

Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll

Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.

Zoom

Commands for zooming the window, in other words, changing the time scale. Choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +

Zoom Out zooms out on the time scale. Shortcut key: –

10ns, 100ns, 1 μ s, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

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

10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Power Log

A heading that shows that the Power Log-specific commands below are available.

Enable

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the window. If no data has been collected for a graph, **no data** will appear instead of the graph.

Clear

Deletes the log information. Note that this will happen also when you reset the debugger.

Viewing Range

Displays a dialog box, see *Viewing Range dialog box*, page 194.

Size

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

Solid Graph

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

Show Numerical Value

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

Go To Source

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

Open Setup Window

Opens the **Power Log Setup** window.

Select Graphs

Selects which graphs to be displayed in the **Timeline** window.

Time Axis Unit

Selects the unit used in the time axis; choose between **Seconds** and **Cycles**.

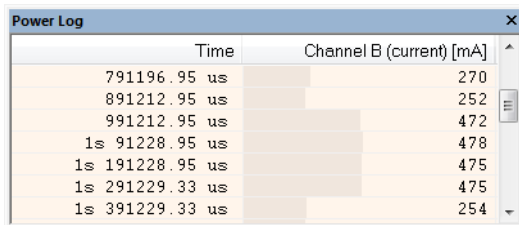
If **Cycles** is not available, the graphs are based on different clock sources. In that case you can view cycle values as tooltip information by pointing at the graph with your mouse pointer.

Profile Selection

Enables profiling time intervals in the **Function Profiler** window. Note that this command is only available if the C-SPY driver supports PC Sampling.

Power Log window

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



Time	Channel B (current) [mA]
791196.95 us	270
891212.95 us	252
991212.95 us	472
1s 91228.95 us	478
1s 191228.95 us	475
1s 291229.33 us	475
1s 391229.33 us	254

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.

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

See also *Debugging in the power domain*, page 217.

Requirements

The C-SPY Power Debugger driver and the Power Debugger probe.

Display area

This area contains these columns:

Time

The time from the application reset until the event. Note that the time offset of a continuous block of samples (in other words when you start and stop the execution) is approximated by IAR Embedded Workbench. However, relative time periods within a block are very accurate because each sample is timestamped at the debug probe, based on an on-board clock.

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.

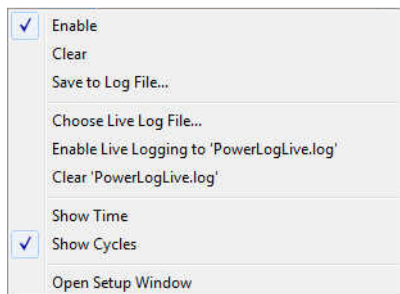
This column is available when you have selected **Show Cycles** from the context menu.

Name [unit]

The power measurement value expressed in the unit you specified in the **Power Setup** window.

Context menu

This context menu is available:



These commands are available:

Enable

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

Clear

Deletes the log information. Note that this will also happen when you reset the debugger, or if you change the execution frequency in the **Power Log Setup** 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.

Choose Live Log File

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

Enable Live Logging to

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

Clear *log file*

Clears the content of the live log file.

Show Time

Displays the **Time** column.

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

Show Cycles

Displays the **Cycles** column.

If the **Cycles** column is not supported in the C-SPY driver you are using, this menu command is not available.

Open Setup Window

Opens the **Power Log Setup** window.

The format of the log file

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

Time/Cycles

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

Approx

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

PC

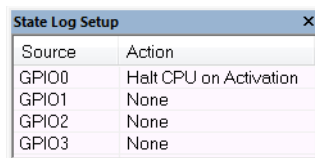
The value of the program counter close to the point where the power value was logged.

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.

State Log Setup window

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



Source	Action
GPIO0	Halt CPU on Activation
GPIO1	None
GPIO2	None
GPIO3	None

Use the **State Log Setup** window to specify whether to halt the CPU in connection with activity—state changes—for the GPIO input pins.

Requirements

The C-SPY Power Debugger driver and the Power Debugger probe.

Display area

Each row in this area displays how the CPU behaves for the listed source (GPIO input pins).

Source

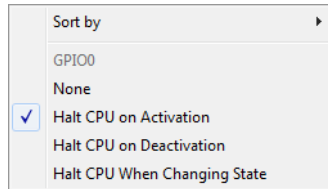
The name of the GPIO input pin.

Action

Specifies an action to take. To choose an action, right-click on the row to open the context menu.

Context menu

This context menu is available:



These commands are available:

Sort by

Commands for sorting the window contents. Choose between:

ID sorts the sources by their ID.

Name sorts the sources alphabetically.

None

The CPU keeps executing when the specified source becomes active, is deactivated, or changes state.

Halt CPU on Activation

The execution stops when the specified source becomes active. This might take a few execution cycles.

Halt CPU on Deactivation

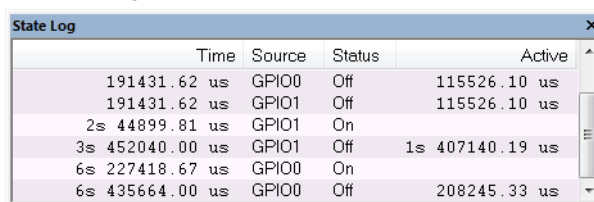
The execution stops when the specified source is deactivated. This might take a few execution cycles.

Halt CPU When Changing State

The execution stops when the specified source changes state. This might take a few execution cycles.

State Log window

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



Time	Source	Status	Active
191431.62 us	GPIO0	Off	115526.10 us
191431.62 us	GPIO1	Off	115526.10 us
2s 44899.81 us	GPIO1	On	
3s 452040.00 us	GPIO1	Off	1s 407140.19 us
6s 227418.67 us	GPIO0	On	
6s 435664.00 us	GPIO0	Off	208245.33 us

This window logs activity—state changes—for GPIO input pins.

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.

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

See also *Displaying a power profile and analyzing the result*, page 217 and *Timeline window—State Log graph*, page 235.

Requirements

The C-SPY Power Debugger driver and the Power Debugger probe.

Display area

This area contains these columns:

Time

The time from the application reset until the event. Note that the time offset of a continuous block of samples (in other words when you start and stop the execution) is approximated by IAR Embedded Workbench. However, relative time periods within a block are very accurate because each sample is timestamped at the debug probe, based on an on-board clock.

This column is available when you have selected **Show Time** from the context menu. If the **Show Time** command is not available, the **Time** column is displayed by default.

Cycles

The number of cycles from the start of the execution until the event.

A cycle count displayed in italics indicates an approximative value. Italics is used when the target system has not been able to collect a correct value, but instead had to approximate it.

This column is available when you have selected **Show Cycles** from the context menu provided that the C-SPY driver you are using supports it.

Source

The name of the GPIO input pin.

Status

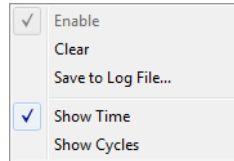
The status at the given time.

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.

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 happen also 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 always 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 C-SPY driver you are using does not support the **Cycles** column, this menu command is not available.

State Log Summary window

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

Source	Count	First Time	Total (Time)	Total (%)	Shortest	Longest	Min Interval	Max Interval
GPIO0	2	76721.90 us	453404.95 us	-	159430.86 us	293974.10 us	343331.05 us	343331.05 us
GPIO1	4	76721.90 us	400127.24 us	-	60.95 us	155872.00 us	87100.57 us	203318.48 us

This window displays a summary of logged activity—state changes—GPIO input pins.

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 217 and *Timeline window—State Log graph*, page 235.

Requirements

The C-SPY Power Debugger driver and the Power Debugger probe.

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 GPIO input pin.

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.

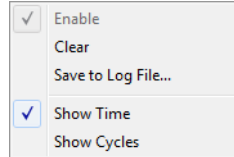
Current time

The current time or cycles—execution time since the start of execution or the number of cycles. (This information might not be available in the C-SPY driver you are using.)

** 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 happen also 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 always displayed by default in the C-SPY driver you are using, this menu command is not available.

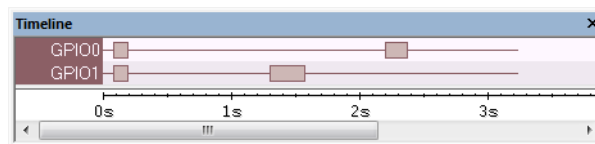
Show Cycles

Displays the **Cycles** column.

If the C-SPY driver you are using does not support the **Cycles** column, 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 the GPIO pins in relation to a common time axis.

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

For more information about the **Timeline** window, how to display a graph, and the other supported graphs, see *Reference information on application timeline*, page 180.

See also, *Requirements and restrictions for power debugging*, page 213.

Requirements

The C-SPY Power Debugger driver and the Power Debugger probe.

Display area

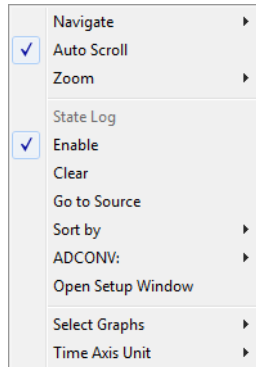
Where:

- 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 GPIO pins 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 231.
- If the bar is displayed without a vertical border, the missing border indicates an approximate time for the log.

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:



Note: The exact contents of the context menu you see on the screen depends on which features that your combination of software and hardware supports. However, the list of menu commands below is complete and covers all possible commands.

These commands are available:

Navigate

Commands for navigating the graph(s). Choose between:

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

Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll

Toggles automatic scrolling on or off. When on, the most recently collected data is automatically displayed when you choose **Navigate>End**.

Zoom

Commands for zooming the window, in other words, changing the time scale. Choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +

Zoom Out zooms out on the time scale. Shortcut key: –

10ns, 100ns, 1us, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

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

10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

State Log

A heading that shows that the State 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 happen also 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.

Open Setup Window

Opens the **State Log Setup** window. This command might not be supported by the combination of C-SPY driver and debug probe you are using.

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.

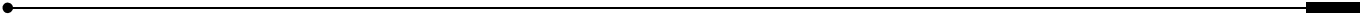
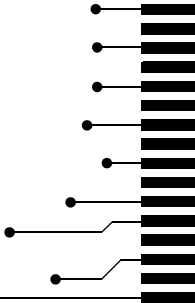
Profile Selection

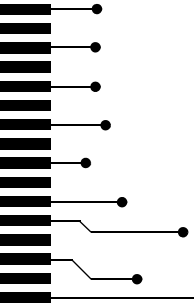
Enables profiling time intervals in the **Function Profiler** window. Note that this command is only available if the C-SPY driver supports PC Sampling.

Part 3. Advanced debugging

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

- Interrupts
- C-SPY macros
- The C-SPY command line utility—`cspybat`





Interrupts

- Introduction to interrupts
- Using the interrupt system
- Reference information on interrupts

Introduction to interrupts

These topics are covered:

- Briefly about the interrupt simulation system
- Interrupt characteristics
- C-SPY system macros for interrupt simulation
- Target-adapting the interrupt simulation system

See also:

- *Reference information on C-SPY system macros*, page 265
- *Breakpoints*, page 105
- *The IAR C/C++ Compiler Reference Guide for AVR*

BRIEFLY ABOUT THE INTERRUPT SIMULATION SYSTEM

By simulating interrupts, you can test the logic of your interrupt service routines and debug the interrupt handling in the target system long before any hardware is available. If you use simulated interrupts in conjunction with C-SPY macros and breakpoints, you can compose a complex simulation of, for instance, interrupt-driven peripheral devices.

The C-SPY Simulator includes an interrupt simulation system where you can simulate the execution of interrupts during debugging. You can configure the interrupt simulation system so that it resembles your hardware interrupt system.

The interrupt system has the following features:

- Simulated interrupt support for the AVR microcontroller
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for various devices
- Configuration of hold time, probability, and timing variation
- State information for locating timing problems

- Configuration of interrupts using a dialog box or a C-SPY system macro—that is, one interactive and one automating interface.

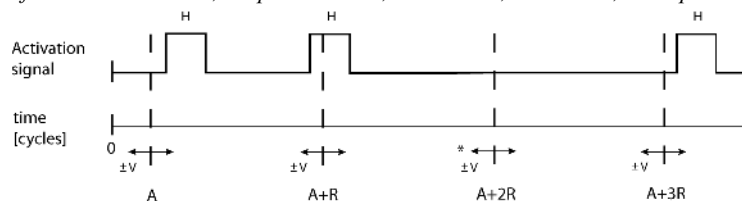
All interrupts you define using the **Interrupts** dialog box exist only until they have been serviced and are not preserved between sessions.



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 **Interrupts** dialog box or a system macro.

INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability*.



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

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

The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation time. However, interrupts can only be raised between instructions, which means that a full assembler instruction must have been executed before the interrupt is generated, regardless of how many cycles an instruction takes.

To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—the probability, in percent, that the interrupt will actually appear in a period—and *variance*—a time variation range as a percentage of the repeat interval. These options make it possible to randomize the interrupt simulation. You can also specify a *hold time* which describes how long the interrupt remains pending until removed if it has not been processed.

C-SPY SYSTEM MACROS FOR INTERRUPT SIMULATION

Macros are useful when you already have sorted out the details of the simulated interrupt so that it fully meets your requirements. If you write a macro function containing definitions for the simulated interrupts, you can execute the functions automatically when C-SPY starts. Another advantage is that your simulated interrupt definitions will be documented if you use macro files, and if you are several engineers involved in the development project you can share the macro files within the group.

The C-SPY Simulator provides these predefined system macros related to interrupts:

```
__enableInterrupts
__disableInterrupts
__orderInterrupt
__cancelInterrupt
__cancelAllInterrupts
```

The parameters of the first five macros correspond to the equivalent entries of the **Interrupts** dialog box.

For more information about each macro, see *Reference information on C-SPY system macros*, page 265.

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 a simplified behavior compared to the hardware. This means that the execution of an interrupt is only dependent on the status of the global interrupt enable bit.

To simulate device-specific interrupts, the interrupt system must have detailed information about each available interrupt. Except for default settings, this information is provided in the device description files. The default settings are used if no device description file has been specified.

For information about device description files, see *Selecting a device description file*, page 51.

Using the interrupt system

These tasks are covered:

- Simulating a simple interrupt

See also:

- *Using C-SPY macros*, page 253 for details about how to use a setup file to define simulated interrupts at C-SPY startup
- The tutorial *Simulating an interrupt* in the Information Center.

SIMULATING A SIMPLE INTERRUPT

This example demonstrates the method for simulating a timer interrupt. However, the procedure can also be used for other types of interrupts.

To simulate and debug an interrupt:

- 1 Assume this simple application which contains an interrupt service routine for a timer, which increments a tick variable. The main function sets the necessary status registers. The application exits when 100 interrupts have been generated.

```
#include <sdtio.h>
#include <iom128.h>
#include <intrinsics.h>

volatile int ticks = 0;
void main (void)
{
    /* Add your timer setup code here */

    __enable_interrupt();          /* Enable interrupts */

    while (ticks < 100);          /* Endless loop */
    printf("Done\n");
}

/* Timer interrupt service routine */
#pragma vector = TIMERO_COMP_vect
__interrupt void basic_timer(void)
{
    ticks += 1;
}
```

- 2 Add the file to your project.
- 3 Choose **Project>Options>General Options>Device** and select **Atmega128**. A matching device description file will automatically be used.

- 4 Build your project and start the simulator.
- 5 Choose **Simulator>Interrupts** to open the **Interrupts** dialog box. Select the **Enable simulation** option to enable interrupt simulation. In the **Interrupt** drop-down list, select the `TIMER0_COMP` interrupt, and verify these settings:

Option	Settings
Activation	4000
Repeat interval	2000
Hold time	10
Probability (%)	100
Variance (%)	0

Table 11: Timer interrupt settings

Click **Install** and then click **OK**.

- 6 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:
 - Generate an interrupt when the cycle counter has passed 4000
 - Continuously repeat the interrupt after approximately 2000 cycles.

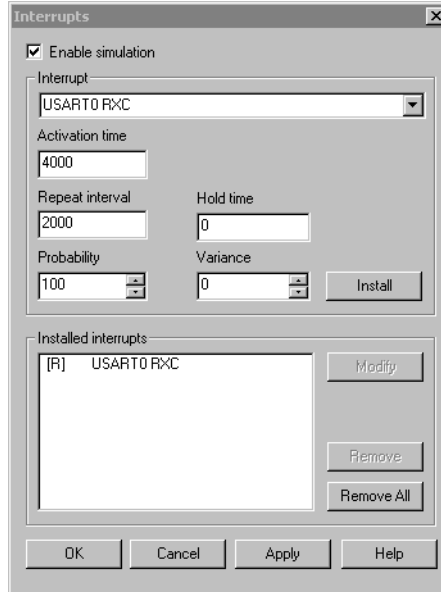
Reference information on interrupts

Reference information about:

- *Interrupts dialog box*, page 248

Interrupts dialog box

The **Interrupts** dialog box is available by choosing **Simulator>Interrupts**.



This dialog box lists all defined interrupts. Use this dialog box to enable or disable the interrupt simulation system, as well as to enable or disable individual interrupts.

Requirements

The C-SPY simulator.

Enable simulation

Enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts are generated.

Interrupt

Lists all available interrupts. Your selection will automatically update the Description box.

The list is populated with entries from the device description file that you have selected.

Activation time

Specify the value of the cycle counter, after which the specified type of interrupt will be generated.

Repeat interval

Specify the periodicity of the interrupt in cycles.

Hold time

Specify how long, in cycles, the interrupt remains pending until removed if it has not been processed.

Probability

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

Variance

Specify a timing variation range, as a percentage of the repeat interval in which the interrupt may 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.

Installed interrupts

Lists the installed interrupts. The interrupt specification text in the list is prefixed with either [S] for a single-shot interrupt or [R] for a repeated interrupt. If the interrupt is activated but pending an additional [P] will be inserted.

Buttons

These buttons are available:

Install	Installs the interrupt you specified.
Modify	Edits an existing interrupt.
Remove	Removes the selected interrupt.
Remove all	Removes all installed interrupts.

C-SPY macros

- Introduction to C-SPY macros
- Using C-SPY macros
- Reference information on the macro language
- Reference information on reserved setup macro function names
- Reference information on C-SPY system macros
- Graphical environment for macros

Introduction to C-SPY macros

These topics are covered:

- Reasons for using C-SPY macros
- Briefly about using C-SPY macros
- Briefly about setup macro functions and files
- Briefly about the macro language

REASONS FOR USING C-SPY MACROS

You can use C-SPY macros either by themselves or in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. Some examples where macros can be useful:

- Automating the debug session, for instance with trace printouts, printing values of variables, and setting breakpoints.
- Hardware configuring, such as initializing hardware registers.
- Feeding your application with simulated data during runtime.
- Simulating peripheral devices, see the chapter *Interrupts*. This only applies if you are using the simulator driver.
- Developing small debug utility functions, for instance calculating the stack depth, see the provided example `StackChk.mac` located in the directory `\AVR\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 263.

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 258.

Example

Consider this example of a macro function which illustrates the various components of the macro language:

```
__var oldVal;
CheckLatest(val)
{
    if (oldVal != val)
    {
        __message "Message: Changed from ", oldVal, " to ", val, "\n";
        oldVal = val;
    }
}
```

Note: Reserved macro words begin with double underscores to prevent name conflicts.

Using C-SPY macros

These tasks are covered:

- Registering C-SPY macros—an overview
- Executing C-SPY macros—an overview
- Registering and executing using setup macros and setup files
- Executing macros using Quick Watch
- Executing a macro by connecting it to a breakpoint
- Aborting a C-SPY macro

For more examples using C-SPY macros, see:

- The tutorial about simulating an interrupt, which you can find in the Information Center
- *Initializing target hardware before C-SPY starts*, page 55.

REGISTERING C-SPY MACROS—AN OVERVIEW

C-SPY must know that you intend to use your defined macro functions, and thus you must *register* your macros. There are various ways to register macro functions:

- You can register macro functions during the C-SPY startup sequence, see *Registering and executing using setup macros and setup files*, page 255.
- You can register macros interactively in the **Macro Registration** window, see *Macro Registration window*, page 302. Registered macros appear in the **Debugger Macros** window, see *Debugger Macros window*, page 304.
- 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 285.

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 255.
- 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 255.
- The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is more specified on designed for C-SPY macros. See *Macro Quicklaunch window*, page 306.
- 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 256.

Which method you choose depends on which level of interaction or automation you want, and depending on at which stage you want to execute your macro.

REGISTERING AND EXECUTING USING SETUP MACROS AND SETUP FILES

It can be convenient to register a macro file during the C-SPY startup sequence. To do this, specify a macro file which you load before starting the debug session. Your macro functions will be automatically registered each time you start the debugger.

If you use the reserved setup macro function names to define the macro functions, you can define exactly at which stage you want the macro function to be executed.

To define a setup macro function and load it during C-SPY startup:

- 1 Create a new text file where you can define your macro function.

For example:

```
execUserSetup()
{
    ...
    __registerMacroFile("MyMacroUtils.mac");
    __registerMacroFile("MyDeviceSimulation.mac");
}
```

This macro function registers the additional macro files `MyMacroUtils.mac` and `MyDeviceSimulation.mac`. Because the macro function is defined with the function name `execUserSetup`, it will be executed directly after your application has been downloaded.

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

The macros will now be registered during the C-SPY startup sequence.

EXECUTING MACROS USING QUICK WATCH

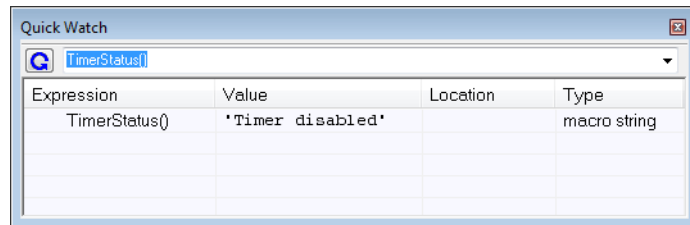
The **Quick Watch** window lets you dynamically choose when to execute a macro function.

- 1 Consider this simple macro function that checks the status of a timer enable bit:

```
TimerStatus()
{
    if ((TimerStatreg & 0x01) != 0) /* Checks the status of reg */
        return "Timer enabled"; /* C-SPY macro string used */
    else
        return "Timer disabled"; /* C-SPY macro string used */
}
```

- 2 Save the macro function using the filename extension `mac`.
- 3 To load the macro file, choose **View>Macros>Macro Registration**. The **Macro Registration** window is displayed. Click **Add** and locate the file using the file browser. The macro file appears in the list of macros in the **Macro Registration** window.
- 4 Select the macro you want to register and your macro will appear in the **Debugger Macros** window.
- 5 Choose **View>Quick Watch** to open the **Quick Watch** window, type the macro call `TimerStatus()` in the text field and press Return,

Alternatively, in the macro file editor window, select the macro function name `TimerStatus()`. Right-click, and choose **Quick Watch** from the context menu that appears.



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

For more information, see *Quick Watch window*, page 100.

EXECUTING A MACRO BY CONNECTING IT TO A BREAKPOINT

You can connect a macro to a breakpoint. The macro will then be executed when the breakpoint is triggered. The advantage is that you can stop the execution at locations of particular interest and perform specific actions there.



For instance, you can easily produce log reports containing information such as how the values of variables, symbols, or registers change. To do this you might set a breakpoint on a suspicious location and connect a log macro to the breakpoint. After the execution you can study how the values of the registers have changed.

To create a log macro and connect it to a breakpoint:

- 1 Assume this skeleton of a C function in your application source code:

```
int fact(int x)
{
    ...
}
```


- 2 Create a simple log macro function like this example:

```
logfact ()
{
  __message "fact (" ,x, " )";
}
```

The `__message` statement will log messages to the Log window.

Save the macro function in a macro file, with the filename extension `mac`.

- 3 To register the macro, choose **View>Macros>Macro Registration** to open the **Macro Registration** window and add your macro file to the list. Select the file to register it. Your macro function will appear in the **Debugger Macros** window.
- 4 To set a code breakpoint, click the **Toggle Breakpoint** button on the first statement within the function `fact` in your application source code. Choose **View>Breakpoints** to open the **Breakpoints** window. Select your breakpoint in the list of breakpoints and choose the **Edit** command from the context menu.
- 5 To connect the log macro function to the breakpoint, type the name of the macro function, `logfact ()`, in the **Action** field and click **Apply**. Close the dialog box.
- 6 Execute your application source code. When the breakpoint is triggered, the macro function will be executed. You can see the result in the **Log** window.
 - Note that the expression in the **Action** field is evaluated only when the breakpoint causes the execution to really stop. If you want to log a value and then automatically continue execution, you can either:
 - Use a **Log** breakpoint, see *Log breakpoints dialog box*, page 121
 - Use the **Condition** field instead of the **Action** field. For an example, see *Performing a task and continuing execution*, page 116.
- 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 261.

For an example where a serial port input buffer is simulated using the method of connecting a macro to a breakpoint, see the tutorial *Simulating an interrupt* in the Information Center.

ABORTING A C-SPY MACRO

To abort a C-SPY macro:

- 1 Press **Ctrl+Shift+.** (period) for a short while.
- 2 A message that says that the macro has terminated is displayed in the **Debug Log** window.

This method can be used if you suspect that something is wrong with the execution, for example because it seems not to terminate in a reasonable time.

Reference information on the macro language

Reference information about:

- *Macro functions*, page 258
- *Macro variables*, page 258
- *Macro parameters*, page 259
- *Macro strings*, page 259
- *Macro statements*, page 260
- *Formatted output*, page 261.

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 86.

The syntax for defining one or more macro variables is:

```
__var nameList;
```

where *nameList* is a list of C-SPY variable names separated by commas.

A macro variable defined outside a macro body has global scope, and it exists throughout the whole debugging session. A macro variable defined within a macro body is created when its definition is executed and destroyed on return from the macro.

By default, macro variables are treated as signed integers and initialized to 0. When a C-SPY variable is assigned a value in an expression, it also acquires the type of that expression. For example:

Expression	What it means
<code>myvar = 3.5;</code>	<code>myvar</code> is now type <code>double</code> , value <code>3.5</code> .
<code>myvar = (int*)i;</code>	<code>myvar</code> is now type <code>pointer to int</code> , and the value is the same as <code>i</code> .

Table 12: Examples of C-SPY macro variables

In case of a name conflict between a C symbol and a C-SPY macro variable, C-SPY macro variables have a higher precedence than C variables. Note that macro variables are allocated on the debugger host and do not affect your application.

MACRO PARAMETERS

A macro parameter is intended for parameterization of device support. The named parameter will behave as a normal C-SPY macro variable with these differences:

- The parameter definition can have an initializer
- Values of a parameters can be set through options (either in the IDE or in `cspybat`).
- A value set from an option will take precedence over a value set by an initializer
- A parameter must have an initializer, be set through an option, or both. Otherwise, it has an undefined value, and accessing it will cause a runtime error.

The syntax for defining one or more macro parameters is:

```
__param param[ = value, ...;]
```

Use the command line option `--macro_param` to specify a value to a parameter, see `--macro_param`, page 330.

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 261.

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 86.

Conditional statements

```
if (expression)
    statement
```

```
if (expression)
    statement
else
    statement
```

Loop statements

```
for (init_expression; cond_expression; update_expression)
    statement
```

```
while (expression)
    statement
```

```
do
    statement
while (expression);
```

Return statements

```
return;
```

```
return expression;
```

If the return value is not explicitly set, signed int 0 is returned by default.

Blocks

Statements can be grouped in blocks.

```
{
    statement1
    statement2
    .
    .
    statementN
}
```

FORMATTED OUTPUT

C-SPY provides various methods for producing formatted output:

<code>__message <i>argList</i>;</code>	Prints the output to the Debug Log window.
<code>__fmessage <i>file</i>, <i>argList</i>;</code>	Prints the output to the designated file.
<code>__smessage <i>argList</i>;</code>	Returns a string containing the formatted output.

where *argList* is a comma-separated list of C-SPY expressions or strings, and *file* is the result of the `__openFile` system macro, see `__openFile`, page 280.

To produce messages in the **Debug Log** window:

```
var1 = 42;
var2 = 37;
__message "This line prints the values ", var1, " and ", var2,
" in the Log window.";
```

This produces this message in the Log window:

This line prints the values 42 and 37 in the Log window.

To write the output to a designated file:

```
__fmessage myfile, "Result is ", res, "!\n";
```

To produce strings:

```
myMacroVar = __smessage 42, " is the answer.";
myMacroVar now contains the string "42 is the answer.".
```

Specifying display format of arguments

To override the default display format of a scalar argument (number or pointer) in *argList*, suffix it with a `:` followed by a format specifier. Available specifiers are:

<code>%b</code>	for binary scalar arguments
<code>%o</code>	for octal scalar arguments
<code>%d</code>	for decimal scalar arguments
<code>%x</code>	for hexadecimal scalar arguments
<code>%c</code>	for character scalar arguments

These match the formats available in the **Watch** and **Locals** windows, but number prefixes and quotes around strings and characters are not printed. Another example:

```
__message "The character '", cvar:%c, "' has the decimal value
", cvar;
```

Depending on the value of the variables, this produces this message:

The character 'A' has the decimal value 65

Note: A character enclosed in single quotes (a character literal) is an integer constant and is not automatically formatted as a character. For example:

```
__message 'A', " is the numeric value of the character ",
'A':%c;
```

would produce:

```
65 is the numeric value of the character A
```

Note: The default format for certain types is primarily designed to be useful in the **Watch** window and other related windows. For example, a value of type `char` is formatted as `'A' (0x41)`, while a pointer to a character (potentially a C string) is formatted as `0x8102 "Hello"`, where the string part shows the beginning of the string (currently up to 60 characters).

When printing a value of type `char*`, use the `%x` format specifier to print just the pointer value in hexadecimal notation, or use the system macro `__toString` to get the full string value.

Reference information on reserved setup macro function names

There are reserved setup macro function names that you can use for defining your setup macro functions. By using these reserved names, your function will be executed at defined stages during execution. For more information, see *Briefly about setup macro functions and files*, page 252.

Reference information about:

- `execUserPreload`
- `execUserExecutionStarted`
- `execUserExecutionStopped`
- `execUserSetup`
- `execUserPreReset`
- `execUserReset`
- `execUserExit`

execUserPreload

Syntax	<code>execUserPreload</code>
For use with	All C-SPY drivers.
Description	Called after communication with the target system is established but before downloading the target application. Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.

execUserExecutionStarted

Syntax	<code>execUserExecutionStarted</code>
For use with	All C-SPY drivers.
Description	Called when the debugger is about to start or resume execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window.

execUserExecutionStopped

Syntax	<code>execUserExecutionStopped</code>
For use with	All C-SPY drivers.
Description	Called when the debugger has stopped execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window.

execUserSetup

Syntax	<code>execUserSetup</code>
For use with	All C-SPY drivers.
Description	Called once after the target application is downloaded. Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.



If you define interrupts or breakpoints in a macro file that is executed at system start (using `execUserSetup`) we strongly recommend that you also make sure that they are removed at system shutdown (using `execUserExit`). An example is available in `SetupSimple.mac`, see the tutorials in the Information Center.

The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time `execUserSetup` is executed again. This seriously affects the execution speed.

execUserPreReset

Syntax	<code>execUserPreReset</code>
For use with	All C-SPY drivers.
Description	Called each time just before the reset command is issued. Implement this macro to set up any required device state.

execUserReset

Syntax	<code>execUserReset</code>
For use with	All C-SPY drivers.
Description	Called each time just after the reset command is issued. Implement this macro to set up and restore data.

execUserExit

Syntax	<code>execUserExit</code>
For use with	All C-SPY drivers.
Description	Called once when the debug session ends. Implement this macro to save status data etc.

Reference information on C-SPY system macros

This section gives reference information about each of the C-SPY system macros.

This table summarizes the pre-defined system macros:

Macro	Description
<code>__abortLaunch</code>	Aborts the launch of the debugger
<code>__cancelAllInterrupts</code>	Cancels all ordered interrupts
<code>__cancelInterrupt</code>	Cancels an interrupt
<code>__clearBreak</code>	Clears a breakpoint

Table 13: Summary of system macros

Macro	Description
<code>__closeFile</code>	Closes a file that was opened by <code>__openFile</code>
<code>__delay</code>	Delays execution
<code>__disableInterrupts</code>	Disables generation of interrupts
<code>__driverType</code>	Verifies the driver type
<code>__enableInterrupts</code>	Enables generation of interrupts
<code>__evaluate</code>	Interprets the input string as an expression and evaluates it.
<code>__fillMemory8</code>	Fills a specified memory area with a byte value.
<code>__fillMemory16</code>	Fills a specified memory area with a 2-byte value.
<code>__fillMemory32</code>	Fills a specified memory area with a 4-byte value.
<code>__getCycleCounter</code>	Reads the cycle counter
<code>__isBatchMode</code>	Checks if C-SPY is running in batch mode or not.
<code>__loadImage</code>	Loads an image.
<code>__memoryRestore</code>	Restores the contents of a file to a specified memory zone
<code>__memoryRestoreFromFile</code>	Reads from a file and restores to memory
<code>__memorySave</code>	Saves the contents of a specified memory area to a file
<code>__memorySaveToFile</code>	Saves a range of a memory zone to a file
<code>__messageBoxYesCancel</code>	Displays a Yes/Cancel dialog box for user interaction
<code>__messageBoxYesNo</code>	Displays a Yes/No dialog box for user interaction
<code>__openFile</code>	Opens a file for I/O operations
<code>__orderInterrupt</code>	Generates an interrupt
<code>__readFile</code>	Reads from the specified file
<code>__readFileByte</code>	Reads one byte from the specified file
<code>__readMemory8,</code> <code>__readMemoryByte</code>	Reads one byte from the specified memory location
<code>__readMemory16</code>	Reads two bytes from the specified memory location
<code>__readMemory32</code>	Reads four bytes from the specified memory location
<code>__registerMacroFile</code>	Registers macros from the specified file
<code>__resetFile</code>	Rewinds a file opened by <code>__openFile</code>
<code>__setCodeBreak</code>	Sets a code breakpoint
<code>__setComplexBreak</code>	Sets a complex breakpoint

Table 13: Summary of system macros

Macro	Description
<code>__setDataBreak</code>	Sets a data breakpoint
<code>__setLogBreak</code>	Sets a log breakpoint
<code>__setSimBreak</code>	Sets a simulation breakpoint
<code>__setTraceStartBreak</code>	Sets a trace start breakpoint
<code>__setTraceStopBreak</code>	Sets a trace stop breakpoint
<code>__sourcePosition</code>	Returns the file name and source location if the current execution location corresponds to a source location
<code>__strFind</code>	Searches a given string for the occurrence of another string
<code>__subString</code>	Extracts a substring from another string
<code>__targetDebuggerVersion</code>	Returns the version of the target debugger
<code>__toLower</code>	Returns a copy of the parameter string where all the characters have been converted to lower case
<code>__toString</code>	Prints strings
<code>__toUpper</code>	Returns a copy of the parameter string where all the characters have been converted to upper case
<code>__unloadImage</code>	Unloads a debug image
<code>__writeFile</code>	Writes to the specified file
<code>__writeFileByte</code>	Writes one byte to the specified file
<code>__writeMemory8,</code> <code>__writeMemoryByte</code>	Writes one byte to the specified memory location
<code>__writeMemory16</code>	Writes a two-byte word to the specified memory location
<code>__writeMemory32</code>	Writes a four-byte word to the specified memory location

Table 13: Summary of system macros

`__abortLaunch`

Syntax	<code>__abortLaunch(message)</code>
Parameters	<i>message</i> A string that is printed as an error message when the macro executes.
Return value	None.

For use with	All C-SPY drivers.
Description	This macro can be used for aborting a debugger launch, for example if another macro sees that something goes wrong during initialization and cannot perform a proper setup. This is an emergency exit, not a recommended way to end a debug session like the C library function <code>abort()</code> .
Example	<pre>if (!__messageBoxYesCancel("Do you want to mass erase to unlock the device?", "Unlocking device")) { __abortLaunch("Unlock canceled. Debug session cannot continue."); }</pre>

__cancelAllInterrupts

Syntax	<code>__cancelAllInterrupts()</code>
Return value	<code>int 0</code>
For use with	The C-SPY Simulator.
Description	Cancels all ordered interrupts.

__cancelInterrupt

Syntax	<code>__cancelInterrupt(<i>interrupt_id</i>)</code>
Parameters	<i>interrupt_id</i> The value returned by the corresponding <code>__orderInterrupt</code> macro call (unsigned long).

Return value

Result	Value
Successful	<code>int 0</code>
Unsuccessful	Non-zero error number

Table 14: `__cancelInterrupt` return values

For use with	The C-SPY Simulator.
Description	Cancels the specified interrupt.

__clearBreak

Syntax	<code>__clearBreak(<i>break_id</i>)</code>
Parameters	<i>break_id</i> The value returned by any of the set breakpoint macros.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Clears a user-defined breakpoint.
See also	<i>Breakpoints</i> , page 105.

__closeFile

Syntax	<code>__closeFile(<i>fileHandle</i>)</code>
Parameters	<i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Closes a file previously opened by <code>__openFile</code> .

__delay

Syntax	<code>__delay(<i>value</i>)</code>
Parameters	<i>value</i> The number of milliseconds to delay execution.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Delays execution the specified number of milliseconds.

__disableInterrupts

Syntax `__disableInterrupts()`

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 15: `__disableInterrupts` return values

For use with The C-SPY Simulator.

Description Disables the generation of interrupts.

__driverType

Syntax `__driverType(driver_id)`

Parameters

driver_id

A string corresponding to the driver you want to check for. Choose one of these:

"sim" corresponds to the simulator driver.

"jtagicemkII" corresponds to the C-SPY JTAGICE mkII driver

"atmel-ice" corresponds to the C-SPY Atmel-ICE driver

"power debugger" corresponds to the C-SPY Power Debugger driver

"jtagice3" corresponds to the C-SPY JTAGICE3 driver

"avrone" corresponds to the C-SPY AVR ONE! driver

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 16: `__driverType` return values

For use with All C-SPY drivers

Description Checks to see if the current C-SPY driver is identical to the driver type of the *driver_id* parameter.

Example

```
__driverType("sim")
```

If the simulator is the current driver, the value 1 is returned. Otherwise 0 is returned.

__enableInterrupts

Syntax

```
__enableInterrupts()
```

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 17: *__enableInterrupts* return values

For use with

The C-SPY Simulator.

Description

Enables the generation of interrupts.

__evaluate

Syntax

```
__evaluate(string, valuePtr)
```

Parameters

string

Expression string.

valuePtr

Pointer to a macro variable storing the result.

Return value

Result	Value
Successful	int 0
Unsuccessful	int 1

Table 18: *__evaluate* return values

For use with

All C-SPY drivers.

Description

This macro interprets the input string as an expression and evaluates it. The result is stored in a variable pointed to by *valuePtr*.

Example

This example assumes that the variable *i* is defined and has the value 5:

```
__evaluate("i + 3", &myVar)
```

The macro variable `myVar` is assigned the value 8.

__fillMemory8

Syntax

```
__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 134.

length

An integer that specifies how many bytes are affected.

format

One of these alternatives:

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, "DATA", 0x10, "OR");
```

__fillMemory16

Syntax

```
__fillMemory16(value, address, zone, length, format)
```


Parameters	<p><i>value</i> An integer that specifies the value.</p> <p><i>address</i> An integer that specifies the memory start address.</p> <p><i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 134.</p> <p><i>length</i> An integer that defines how many 2-byte entities to be affected.</p> <p><i>format</i> One of these alternatives:</p> <table> <tr> <td>Copy</td> <td><i>value</i> will be copied to the specified memory area.</td> </tr> <tr> <td>AND</td> <td>An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> <tr> <td>OR</td> <td>An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> <tr> <td>XOR</td> <td>An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.</td> </tr> </table>	Copy	<i>value</i> will be copied to the specified memory area.	AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.	OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.	XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
Copy	<i>value</i> will be copied to the specified memory area.								
AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.								
Return value	int 0								
For use with	All C-SPY drivers.								
Description	Fills a specified memory area with a 2-byte value.								
Example	<code>__fillMemory16(0xCDCD, 0x7000, "DATA", 0x200, "Copy");</code>								

__fillMemory32

Syntax	<code>__fillMemory32(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>
Parameters	<p><i>value</i> An integer that specifies the value.</p> <p><i>address</i> An integer that specifies the memory start address.</p>

*zone*A string that specifies the memory zone, see *C-SPY memory zones*, page 134.*length*

An integer that defines how many 4-byte entities to be affected.

format

One of these alternatives:

Copy	<i>value</i> will be copied to the specified memory area.
AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.

Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Fills a specified memory area with a 4-byte value.
Example	<code>__fillMemory32(0x0000FFFF, 0x4000, "DATA", 0x1000, "XOR");</code>

__getCycleCounter

Syntax	<code>__getCycleCounter()</code>
Return value	Returns the current value of the cycle counter as a <code>long long int</code> .
For use with	The C-SPY hardware drivers.
Description	Reads the current value of the cycle counter.

__isBatchMode

Syntax `__isBatchMode()`

Return value

Result	Value
True	int 1
False	int 0

Table 19: `__isBatchMode` return values

For use with All C-SPY drivers.

Description This macro returns True if the debugger is running in batch mode, otherwise it returns False.

__loadImage

Syntax `__loadImage(path, offset, debugInfoOnly)`

Parameters

path

A string that identifies the path to the image to download. The path must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide for AVR*.

offset

An integer that identifies the offset to the destination address for the downloaded image.

debugInfoOnly

A non-zero integer value if no code or data should be downloaded to the target system, which means that C-SPY will only read the debug information from the debug file. Or, 0 (zero) for download.

Return value

Value	Result
Non-zero integer number	A unique module identification.
int 0	Loading failed.

Table 20: `__loadImage` return values

For use with All C-SPY drivers.

Description Loads an image (debug file).

Example 1 Your system consists of a ROM library and an application. The application is your active project, but you have a debug file corresponding to the library. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage("ROMfile", 0x8000, 1);
```

This macro call loads the debug information for the ROM library `ROMfile` without downloading its contents (because it is presumably already in ROM). Then you can debug your application together with the library.

Example 2 Your system consists of a ROM library and an application, but your main concern is the library. The library needs to be programmed into flash memory before a debug session. While you are developing the library, the library project must be the active project in the IDE. In this case you can add this macro call in the `execUserSetup` macro in a C-SPY macro file, which you associate with your project:

```
__loadImage("ApplicationFile", 0x8000, 0);
```

The macro call loads the debug information for the application and downloads its contents (presumably into RAM). Then you can debug your library together with the application.

See also *Images*, page 340 and *Loading multiple images*, page 53.

__memoryRestore

Syntax

```
__memoryRestore(zone, filename)
```

Parameters

zone

A string that specifies the memory zone, see *C-SPY memory zones*, page 134.

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 AVR*.

Return value

0 if successful, otherwise 1

For use with

All C-SPY drivers.

Description

Reads the contents of a file and saves it to the specified memory zone. It is recommended that you use this macro instead of `__memoryRestoreFromFile`.

Example `__memoryRestore("DATA", "c:\\temp\\saved_memory.hex");`

See also *Memory Restore dialog box*, page 144.

__memoryRestoreFromFile

Syntax `__memoryRestoreFromFile(filename, zone)`

Parameters

<i>filename</i>	The file to be read.
<i>zone</i>	The memory zone name (string); for a list of available zones, see <i>C-SPY memory zones</i> , page 134.

Return value 0 if successful, otherwise 1

For use with All C-SPY drivers.

Description Reads the contents of a file in intel-hex or Motorola S-record format and writes it to the specified memory zone. This macro is available for backwards compatibility.

Example `__memoryRestoreFromFile("C:\\temp\\tmp.hex", "DATA");`

__memorySave

Syntax `__memorySave(start, stop, format, filename)`

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`

	<code>motorola-s37.</code>
	<i>filename</i>
	A string that specifies the file to write to. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide for AVR</i> .
Return value	0 if successful. At failure, macro execution is aborted and log messages are produced.
For use with	All C-SPY drivers.
Description	Saves the contents of a specified memory area to a file. It is recommended that you use this macro instead of <code>__memorySaveToFile</code> .
Example	<code>__memorySave("DATA:0x00", "DATA:0xFF", "intel-extended", "c:\\temp\\saved_memory.hex");</code>
See also	<i>Memory Save dialog box</i> , page 143.

__memorySaveToFile

Syntax	<code>__memorySaveToFile(filename, zone, start, stop)</code>
Parameters	
	<i>filename</i> The file to be written.
	<i>zone</i> The memory zone name (string); for a list of available zones, see <i>C-SPY memory zones</i> , page 134.
	<i>start</i> The start address of the memory range to be saved.
	<i>stop</i> The stop address of the memory range to be saved.
Return value	0 if successful, otherwise 1
For use with	All C-SPY drivers.
Description	Saves a range of a memory zone to a file. The file is written in the Intel hex format. This macro is available for backwards compatibility.
Example	<code>__memoryRestoreFromFile("C:\\temp\\tmp.hex", "DATA", "0x1000", "0x1100");</code>

__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 21: __messageBoxYesCancel return values

For use with All C-SPY drivers.

Description

Displays a Yes/Cancel dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

__messageBoxYesNo

Syntax `__messageBoxYesNo (message, caption)`

Parameters *message*

A message that will appear in the message box.

caption

The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 22: __messageBoxYesNo return values

For use with All C-SPY drivers.

Description

Displays a Yes/No dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

__openFile

Syntax

```
__openFile(filename, access)
```

Parameters

filename

The file to be opened. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the *IDE Project Management and Building Guide for AVR*.

access

The access type (string).

These are mandatory but mutually exclusive:

"a" append, new data will be appended at the end of the open file

"r" read (by default in text mode; combine with b for binary mode: rb)

"w" write (by default in text mode; combine with b for binary mode: wb)

These are optional and mutually exclusive:

"b" binary, opens the file in binary mode

"t" ASCII text, opens the file in text mode

This access type is optional:

"+" together with r, w, or a; r+ or w+ is *read* and *write*, while a+ is *read* and *append*

Return value

Result	Value
Successful	The file handle
Unsuccessful	An invalid file handle, which tests as False

Table 23: __openFile return values

For use with

All C-SPY drivers.

Description

Opens a file for I/O operations. The default base directory of this macro is where the currently open project file (*.ewp) is located. The argument to __openFile can specify a location relative to this directory. In addition, you can use argument variables such as \$PROJ_DIR\$ and \$TOOLKIT_DIR\$ in the path argument.

Example

```

__var myFileHandle;          /* The macro variable to contain */
                             /* the file handle */
myFileHandle = __openFile("$PROJ_DIR$\\Debug\\Exe\\test.tst",
"r");
if (myFileHandle)
{
    /* successful opening */
}

```

See also For information about argument variables, see the *IDE Project Management and Building Guide for AVR*.

__orderInterrupt

Syntax `__orderInterrupt(specification, first_activation,
repeat_interval, variance,
hold_time, probability)`

Parameters

specification

The interrupt name (string). 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)

hold_time

The hold time (integer)

probability

The probability in percent (integer between 0 and 100)

Return value

The macro returns an interrupt identifier (unsigned long).

If the syntax of *specification* is incorrect, it returns -1.

For use with

The C-SPY Simulator.

Description

Generates an interrupt.

Example

This example generates a repeating interrupt using an infinite hold time first activated after 4000 cycles:

```
__orderInterrupt( "USART0 TXC", 4000, 2000, 0, 0, 100 );
```

__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 24: `__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	<code>__readFileByte(<i>fileHandle</i>)</code>
Parameters	<i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro.
Return value	-1 upon error or end-of-file, otherwise a value between 0 and 255.
For use with	All C-SPY drivers.
Description	Reads one byte from a file.
Example	<pre> __var byte; while ((byte = __readFileByte(myFileHandle)) != -1) { /* Do something with byte */ } </pre>

__readMemory8, __readMemoryByte

Syntax	<code>__readMemory8(<i>address</i>, <i>zone</i>)</code> <code>__readMemoryByte(<i>address</i>, <i>zone</i>)</code>
Parameters	<i>address</i> The memory address (integer). <i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 134.
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads one byte from a given memory location.
Example	<code>__readMemory8(0x0108, "DATA");</code>

__readMemory16

Syntax	<code>__readMemory16(<i>address</i>, <i>zone</i>)</code>
Parameters	<p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 134.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads a two-byte word from a given memory location.
Example	<code>__readMemory16(0x0108, "DATA");</code>

__readMemory32

Syntax	<code>__readMemory32(<i>address</i>, <i>zone</i>)</code>
Parameters	<p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 134.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads a four-byte word from a given memory location.
Example	<code>__readMemory32(0x0108, "DATA");</code>

__registerMacroFile

Syntax	<code>__registerMacroFile(<i>filename</i>)</code>
Parameters	<p><i>filename</i></p> <p>A file containing the macros to be registered (string). The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide for AVR</i>.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Registers macros from a setup macro file. With this function you can register multiple macro files during C-SPY startup.
Example	<code>__registerMacroFile("c:\\testdir\\macro.mac");</code>
See also	<i>Using C-SPY macros</i> , page 253.

__resetFile

Syntax	<code>__resetFile(<i>fileHandle</i>)</code>
Parameters	<p><i>fileHandle</i></p> <p>A macro variable used as filehandle by the <code>__openFile</code> macro.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Rewinds a file previously opened by <code>__openFile</code> .

__setCodeBreak

Syntax	<code>__setCodeBreak(location, count, condition, cond_type, action)</code>						
Parameters	<p><i>location</i></p> <p>A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see <i>Enter Location dialog box</i>, page 130.</p> <p><i>count</i></p> <p>The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).</p> <p><i>condition</i></p> <p>The breakpoint condition (string).</p> <p><i>cond_type</i></p> <p>The condition type; either "CHANGED" or "TRUE" (string).</p> <p><i>action</i></p> <p>An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.</p>						
Return value	<table border="1"> <thead> <tr> <th>Result</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Successful</td> <td>An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.</td> </tr> <tr> <td>Unsuccessful</td> <td>0</td> </tr> </tbody> </table> <p><i>Table 25: __setCodeBreak return values</i></p>	Result	Value	Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.	Unsuccessful	0
Result	Value						
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.						
Unsuccessful	0						
For use with	All C-SPY drivers.						
Description	Sets a code breakpoint, that is, a breakpoint which is triggered just before the processor fetches an instruction at the specified location.						
Examples	<pre>__setCodeBreak("{D:\\src\\prog.c}.12.9", 3, "d>16", "TRUE", "ActionCode()");</pre> <p>This example sets a code breakpoint on the label <code>main</code> in your source:</p> <pre>__setCodeBreak("main", 0, "1", "TRUE", "");</pre>						
See also	<i>Breakpoints</i> , page 105.						

__setComplexBreak

Syntax

```
__setComplexBreak(control, a_addr, b_addr, access_type,
a_access, b_access, complex_data, c_value, d_value, c_compare,
d_compare, action)
```

Parameters

<i>control</i>	<p>Breakpoint control:</p> <p>ENABLE_A to enable a breakpoint at <i>a_addr</i></p> <p>ENABLE_AB to enable breakpoints at <i>a_addr</i> and <i>b_addr</i></p> <p>RANGE to enable a range breakpoint from <i>a_addr</i> to <i>b_addr</i>.</p>
<i>a_addr</i>	<p>A string with a location description. This can be:</p> <p>A source location on the form <i>{filename}.line.col</i>, for example <i>{D:\src\prog.c}.12.9</i>, although this is not very useful for data breakpoints.</p> <p>An absolute location on the form <i>zone:hexaddress</i> or simply <i>hexaddress</i>, for example <i>Memory:0x42</i></p> <p>An expression whose value designates a location, for example <i>myGlobalVariable</i>.</p>
<i>b_addr</i>	<p>A string with a location description. This can be:</p> <p>A source location on the form <i>{filename}.line.col</i>, for example <i>{D:\src\prog.c}.12.9</i>, although this is not very useful for data breakpoints.</p> <p>An absolute location on the form <i>zone:hexaddress</i> or simply <i>hexaddress</i>, for example <i>Memory:0x42</i>.</p> <p>An expression whose value designates a location, for example <i>myGlobalVariable</i>.</p>
<i>access_type</i>	<p>The memory space:</p> <p>DATA for data memory</p> <p>CODE for code memory</p>
<i>a_access</i>	<p>The memory access type:</p> <p>R for read</p> <p>W for write</p> <p>RW for read/write</p>

<i>b_access</i>	<p>The memory access type:</p> <p>R for read</p> <p>W for write</p> <p>RW for read/write</p>
<i>complex_data</i>	<p>Complex data control:</p> <p>C_COMBINED_WITH_A for enable complex data</p> <p>CD_COMBINED_WITH_AB for C combined with A and D combined with B</p> <p>CD_COMBINED_WITH_A for C and D combined with A</p> <p>NOTCD_COMBINED_WITH_A for not C and D combined with A</p> <p>C_MASKED_WITH_D_COMBINED_WITH_A for C masked with D combined with A</p>
<i>c_value</i>	<p>Single-byte value for comparison with the memory contents of <i>a_addr</i> and <i>b_addr</i>.</p>
<i>d_value</i>	<p>Single-byte value for comparison with the memory contents of <i>a_addr</i> and <i>b_addr</i>.</p>
<i>c_compare</i>	<p>The relationship between <i>c_value</i> and the contents of data memory at <i>a_addr</i> and <i>b_addr</i>:</p> <p>EQ matches when <i>c_value</i> and the data value are equal</p> <p>LE matches when <i>c_value</i> is less than the data value</p> <p>GE matches when <i>c_value</i> is greater than or equal to the data value</p>
<i>d_compare</i>	<p>The relationship between <i>d_value</i> and the contents of data memory at <i>a_addr</i> and <i>b_addr</i>:</p> <p>EQ matches when <i>d_value</i> and the data value are equal</p> <p>LE matches when <i>d_value</i> is less than the data value</p> <p>GE matches when <i>d_value</i> is greater than or equal to the data value</p>
<i>action</i>	<p>An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.</p>

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 26: `__setComplexBreak` return values

For use with

The C-SPY AVR ONE! driver
 The C-SPY Atmel-ICE driver
 The C-SPY JTAGICE3 driver

Description

Sets a complex breakpoint, that is, a breakpoint which is triggered directly after the processor has read or written data at the specified location.

Example

The following example enables one data read/write breakpoint at the address for variable `a` and will break if the memory byte value at address is equal to `0x22`, and it enables one data read breakpoint at the address for variable `b` and will break if the memory byte value at address is greater than or equal to `0x11`. When a break occurs, the macro function `ActionData()` will be called.

```
__var brk;
brk=__setComplexBreak("ENABLE_AB", "a", "b", "DATA", "RW", "R",
"CD_COMBINED_WITH_AB", "0x22", "0x11" "EQ", "GE",
"ActionData()");
...
__clearBreak(brk);
```

See also

Breakpoints, page 105.

`__setDataBreak`

Syntax

```
__setDataBreak(location, count, condition, cond_type, access,
action)
```

Parameters

location

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see *Enter Location dialog box*, page 130.

count

The number of times that a breakpoint condition must be fulfilled before a break occurs (integer).

condition

The breakpoint condition (string).

cond_type

The condition type; either "CHANGED" or "TRUE" (string).

access

The memory access type: "R", for read, "W" for write, or "RW" for read/write.

action

An expression, typically a call to a macro, which is evaluated when the breakpoint is detected.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 27: `__setDataBreak` return values

For use with

The C-SPY Simulator.
 The C-SPY Atmel-ICE driver
 The C-SPY JTAGICE3 driver
 The C-SPY AVR ONE! 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

```
__var brk;
brk = __setDataBreak("DATA:0x4710", 3, "d>6", "TRUE",
    "W", "ActionData()");
...
__clearBreak(brk);
```

See also

Breakpoints, page 105.

__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 130.

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 28: __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 261 and *Breakpoints*, page 105.

__setSimBreak

Syntax

```
__setSimBreak(location, access, action)
```

Parameters

location

A string that defines the data location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address or an absolute location. For more information about the location types, see *Enter Location dialog box*, page 130.

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 29: *__setSimBreak* return values

For use with

The C-SPY Simulator.

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

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

__setTraceStartBreak

Syntax `__setTraceStartBreak(location)`

Parameters *location*

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box*, page 130.

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 30: __setTraceStartBreak return values

For use with The C-SPY Simulator.

Description Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is started.

Example

```

__var startTraceBp;
__var stopTraceBp;

traceOn()
{
    startTraceBp = __setTraceStartBreak
        ("C:\\TEMP\\Utilities.c).23.1");
    stopTraceBp = __setTraceStopBreak
        ("C:\\temp\\Utilities.c).30.1");
}

traceOff()
{
    __clearBreak(startTraceBp);
    __clearBreak(stopTraceBp);
}

```

See also

Breakpoints, page 105.

__setTraceStopBreak

Syntax

```
__setTraceStopBreak(location)
```

Parameters

location

A string that defines the code location of the breakpoint, either a valid C-SPY expression whose value evaluates to a valid address, an absolute location, or a source location. For more information about the location types, see *Enter Location dialog box*, page 130.

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 31: __setTraceStopBreak return values

For use with

The C-SPY Simulator.

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is stopped.

Example

See *__setTraceStartBreak*, page 293.

See also *Breakpoints*, page 105.

__sourcePosition

Syntax `__sourcePosition(linePtr, colPtr)`

Parameters

linePtr
Pointer to the variable storing the line number

colPtr
Pointer to the variable storing the column number

Return value

Result	Value
Successful	Filename string
Unsuccessful	Empty (" ") string

Table 32: *__sourcePosition* return values

For use with All C-SPY drivers.

Description If the current execution location corresponds to a source location, this macro returns the filename as a string. It also sets the value of the variables, pointed to by the parameters, to the line and column numbers of the source location.

__strFind

Syntax `__strFind(macroString, pattern, position)`

Parameters

macroString
A macro string.

pattern
The string pattern to search for

position
The position where to start the search. The first position is 0

Return value The position where the pattern was found or -1 if the string is not found.

For use with All C-SPY drivers.

Description	This macro searches a given string (<i>macroString</i>) for the occurrence of another string (<i>pattern</i>).
Example	<pre>__strFind("Compiler", "pile", 0) = 3 __strFind("Compiler", "foo", 0) = -1</pre>
See also	<i>Macro strings</i> , page 259.

__subString

Syntax	<code>__subString(<i>macroString</i>, <i>position</i>, <i>length</i>)</code>
Parameters	<p><i>macroString</i> A macro string.</p> <p><i>position</i> The start position of the substring. The first position is 0.</p> <p><i>length</i> The length of the substring</p>
Return value	A substring extracted from the given macro string.
For use with	All C-SPY drivers.
Description	This macro extracts a substring from another string (<i>macroString</i>).
Example	<pre>__subString("Compiler", 0, 2) The resulting macro string contains Co. __subString("Compiler", 3, 4) The resulting macro string contains pile.</pre>
See also	<i>Macro strings</i> , page 259.

__targetDebuggerVersion

Syntax	<code>__targetDebuggerVersion()</code>
Return value	A string that represents the version number of the C-SPY debugger processor module.
For use with	All C-SPY drivers.

Description This macro returns the version number of the C-SPY debugger processor module.

Example

```
__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 259.

__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 (<code>char*</code> or <code>char[]</code>) into macro strings.
Example	Assuming your application contains this definition: <pre>char const * hptr = "Hello World!";</pre> this macro call: <pre>__toString(hptr, 5)</pre> would return the macro string containing <code>Hello</code> .
See also	<i>Macro strings</i> , page 259.

__ToUpper

Syntax	<code>__ToUpper(<i>macroString</i>)</code>
Parameters	<i>macroString</i> A macro string.
Return value	The converted string.
For use with	All C-SPY drivers.
Description	This macro returns a copy of the parameter <i>macroString</i> where all the characters have been converted to upper case.
Example	<pre>__ToUpper("string")</pre> The resulting macro string contains <code>STRING</code> .
See also	<i>Macro strings</i> , page 259.

__unloadImage

Syntax	<code>__unloadImage(<i>module_id</i>)</code>
Parameters	<i>module_id</i> An integer which represents a unique module identification, which is retrieved as a return value from the corresponding <code>__loadImage</code> C-SPY macro.

Return value

Value	Result
<i>module_id</i>	A unique module identification (the same as the input parameter).
int 0	The unloading failed.

Table 33: *__unloadImage* return values

For use with

All C-SPY drivers.

Description

Unloads debug information from an already downloaded image.

See also

Loading multiple images, page 53 and *Images*, page 340.

__writeFile

Syntax

`__writeFile(fileHandle, value)`

Parameters

*fileHandle*A macro variable used as filehandle by the `__openFile` macro.*value*

An integer.

Return value

int 0

For use with

All C-SPY drivers.

Description

Prints the integer value in hexadecimal format (with a trailing space) to the file *file*.**Note:** The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.

__writeFileByte

Syntax

`__writeFileByte(fileHandle, value)`

Parameters

*fileHandle*A macro variable used as filehandle by the `__openFile` macro.*value*

An integer.

Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes one byte to the file <i>fileHandle</i> .

__writeMemory8, __writeMemoryByte

Syntax	<code>__writeMemory8(value, address, zone)</code> <code>__writeMemoryByte(value, address, zone)</code>
Parameters	<i>value</i> An integer. <i>address</i> The memory address (integer). <i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 134.
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes one byte to a given memory location.
Example	<code>__writeMemory8(0x2F, 0x8020, "DATA");</code>

__writeMemory16

Syntax	<code>__writeMemory16(value, address, zone)</code>
Parameters	<i>value</i> An integer. <i>address</i> The memory address (integer). <i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 134.
Return value	<code>int 0</code>

For use with	All C-SPY drivers.
Description	Writes two bytes to a given memory location.
Example	<code>__writeMemory16(0x2FFF, 0x8020, "DATA");</code>

__writeMemory32

Syntax	<code>__writeMemory32(value, address, zone)</code>
Parameters	<p><i>value</i> An integer.</p> <p><i>address</i> The memory address (integer).</p> <p><i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 134.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Writes four bytes to a given memory location.
Example	<code>__writeMemory32(0x5555FFFF, 0x8020, "DATA");</code>

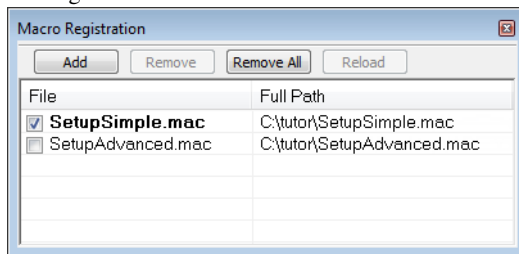
Graphical environment for macros

Reference information about:

- *Macro Registration window*, page 302
- *Debugger Macros window*, page 304
- *Macro Quicklaunch window*, page 306

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 254.

Requirements

None; this window is always available.

Display area

This area contains these columns:

File

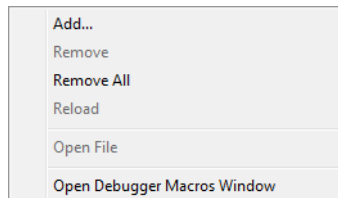
The name of an available macro file. To register the macro file, select the check box to the left of the filename. The name of a registered macro file appears in bold style.

Full path

The path to the location of the added macro file.

Context menu

This context menu is available:



These commands are available:

Add

Opens a file browser where you can locate the macro file that you want to add to the list. This menu command is also available as a function button at the top of the window.

Remove

Removes the selected debugger macro file from the list. This menu command is also available as a function button at the top of the window.

Remove All

Removes all macro files from the list. This menu command is also available as a function button at the top of the window.

Reload

Registers the selected macro file. Typically, this is useful when you have edited a macro file. This menu command is also available as a function button at the top of the window.

Open File

Opens the selected macro file in the editor window.

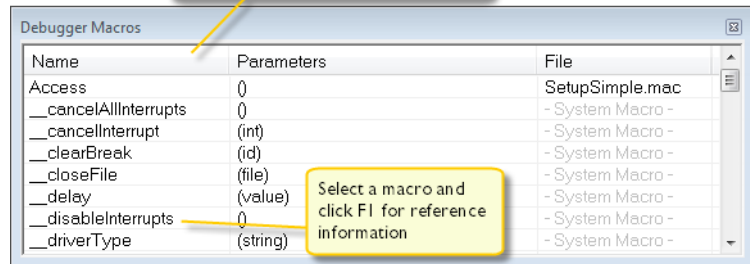
Open Debugger Macros Window

Opens the **Debugger Macros** window.

Debugger Macros window

The **Debugger Macros** window is available from the **View>Macros** submenu during a debug session.

Click the **Name** header or the **File** header to sort alphabetically on either function name or filename.



Select a macro and click F1 for reference information

Use this window to list all registered debugger macro functions, either predefined system macros or your own. This window is useful when you edit your own macro functions and want an overview of all available macros that you can use.

- Click the column headers **Name** or **File** to sort alphabetically on either function name or filename.
- Double-clicking a macro defined in a file opens that file in the editor window.
- To open a macro in the **Macro Quicklaunch** window, drag it from the **Debugger Macros** window and drop it in the **Macro Quicklaunch** window.
- Select a macro and press F1 to get online help information for that macro.

Requirements

None; this window is always available.

Display area

This area contains these columns:

Name

The name of the debugger macro.

Parameters

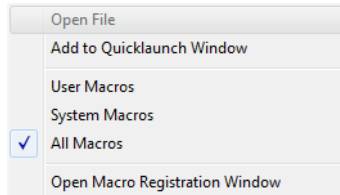
The parameters of the debugger macro.

File

For macros defined in a file, the name of the file is displayed. For predefined system macros, -System Macro- is displayed.

Context menu

This context menu is available:



These commands are available:

Open File

Opens the selected debugger macro file in the editor window.

Add to Quicklaunch Window

Adds the selected macro to the **Macro Quicklaunch** window.

User Macros

Lists only the debugger macros that you have defined yourself.

System Macros

Lists only the predefined system macros.

All Macros

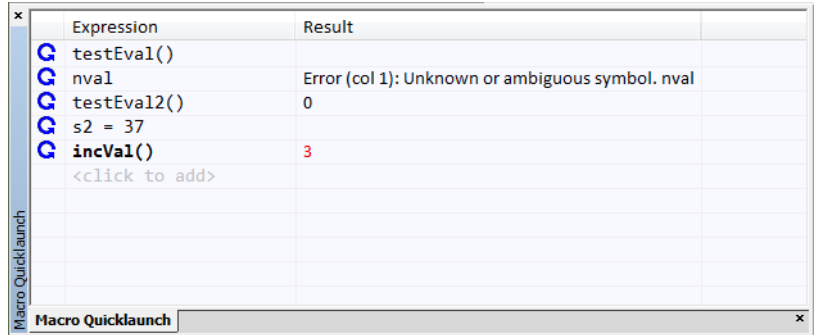
Lists all debugger macros, both predefined system macros and your own.

Open Macro Registration Window

Opens the **Macro Registration** window.

Macro Quicklaunch window

The **Macro Quicklaunch** window is available from the **View** menu.



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 254.

To add an expression:

- I Choose one of these alternatives:
 - Drag the expression to the window
 - In the **Expression** column, type the expression you want to examine.

If the expression you add and want to evaluate is a C-SPY macro, the macro must first be registered, see *Registering C-SPY macros—an overview*, page 254.

To evaluate an expression:



- I Double-click the **Recalculate** icon to calculate the value of that expression.

Requirements

None; this window is always available.

Display area

This area contains these columns:



Recalculate icon

To evaluate the expression, double-click the icon. The latest evaluated expression appears in bold style.

Expression

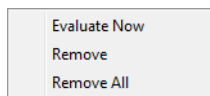
One or several expressions that you want to evaluate. Click <click to add> to add an expression. If the return value has changed since last time, the value will be displayed in red.

Result

Shows the return value from the expression evaluation.

Context menu

This context menu is available:



These commands are available:

Evaluate Now

Evaluates the selected expression.

Remove

Removes the selected expression.

Remove All

Removes all selected expressions.

The C-SPY command line utility—`cspybat`

- Using C-SPY in batch mode
- Summary of C-SPY command line options
- Reference information on C-SPY command line options.

Using C-SPY in batch mode

You can execute C-SPY in batch mode if you use the command line utility `cspybat`, installed in the directory `common\bin`.

These topics are covered:

- Starting `cspybat`
- Output
- Invocation syntax

STARTING CSPYBAT

- 1 To start `cspybat` you must first create a batch file. An easy way to do that is to use one of the batch files that C-SPY automatically generates when you start C-SPY in the IDE.

C-SPY generates a batch file `projectname.buildconfiguration.cspy.bat` every time C-SPY is initialized. In addition, two more files are generated:

- `project.buildconfiguration.general.xml`, which contains options specific to `cspybat`.
- `project.buildconfiguration.driver.xml`, which contains options specific to the C-SPY driver you are using.

You can find the files in the directory `$PROJ_DIR$\settings`. The files contain the same settings as the IDE, and provide hints about additional options that you can use.

- 2 To start `cspybat`, you can use this command line:

```
project.cspybat.bat [debugfile]
```

Note that *debugfile* is optional. You can specify it if you want to use a different debug file than the one that is used in the *project.buildconfiguration.general.xml* file.

OUTPUT

When you run *cspybat*, these types of output can be produced:

- Terminal output from *cspybat* itself

All such terminal output is directed to *stderr*. Note that if you run *cspybat* from the command line without any arguments, the *cspybat* version number and all available options including brief descriptions are directed to *stdout* and displayed on your screen.
- Terminal output from the application you are debugging

All such terminal output is directed to *stdout*, provided that you have used the *--plugin* option. See *--plugin*, page 331.
- Error return codes

cspybat returns status information to the host operating system that can be tested in a batch file. For *successful*, the value *int 0* is returned, and for *unsuccessful* the value *int 1* is returned.

INVOCATION SYNTAX

The invocation syntax for *cspybat* is:

```
cspybat processor_DLL driver_DLL debug_file
      [cspybat_options] --backend driver_options
```

Note: In those cases where a filename is required—including the DLL files—you are recommended to give a full path to the filename.

Parameters

The parameters are:

Parameter	Description
<i>processor_DLL</i>	The processor-specific DLL file; available in <i>avr\bin</i> .
<i>driver_DLL</i>	The C-SPY driver DLL file; available in <i>avr\bin</i> .
<i>debug_file</i>	The object file that you want to debug (filename extension <i>d90</i>). See also <i>--debugfile</i> , page 318.
<i>cspybat_options</i>	The command line options that you want to pass to <i>cspybat</i> . Note that these options are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 315.

Table 34: *cspybat* parameters

Parameter	Description
<code>--backend</code>	Marks the beginning of the parameters to the C-SPY driver; all options that follow will be sent to the driver. Note that this option is mandatory.
<code>driver_options</code>	The command line options that you want to pass to the C-SPY driver. Note that some of these options are mandatory and some are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 315.

Table 34: cspybat parameters (Continued)

Summary of C-SPY command line options

Reference information about:

- General cspybat options
- Options available for all C-SPY drivers
- Options available for the simulator driver
- Options available for all C-SPY hardware debugger drivers
- Options available for the C-SPY Power Debugger driver
- Options available for the C-SPY JTAGICE mkII driver, the C-SPY Dragon driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver
- Options available for the C-SPY JTAGICE mkII driver, the C-SPY Dragon driver, the C-SPY Atmel-ICE driver, the C-SPY Power Debugger driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver
- Options available for the C-SPY JTAGICE mkII driver and the C-SPY Dragon driver
- Options available for the C-SPY Atmel-ICE driver, the C-SPY Power Debugger driver, the C-SPY JTAGICE3 driver, and the C-SPY AVR ONE! driver
- Options available for the C-SPY JTAGICE mkII driver and the C-SPY Dragon driver
- Options available for the C-SPY Dragon driver

GENERAL CSPYBAT OPTIONS

<code>--attach_to_running_target</code>	Makes the debugger attach to a running application at its current location, without resetting the target system.
<code>--backend</code>	Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).

<code>--code_coverage_file</code>	Enables the generation of code coverage information and places it in a specified file.
<code>--cycles</code>	Specifies the maximum number of cycles to run.
<code>--debugfile</code>	Specifies an alternative debug file.
<code>--download_only</code>	Downloads a code image without starting a debug session afterwards.
<code>-f</code>	Extends the command line.
<code>--leave_target_running</code>	Makes the debugger leave the application running on the target hardware after the debug session is closed.
<code>--macro</code>	Specifies a macro file to be used.
<code>--macro_param</code>	Assigns a value to a C-SPY macro parameter.
<code>--plugin</code>	Specifies a plugin file to be used.
<code>--silent</code>	Omits the sign-on message.
<code>--timeout</code>	Limits the maximum allowed execution time.

OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

<code>--64bit_doubles</code>	Specifies that 64-bit doubles are used instead of 32-bit doubles.
<code>--64k_flash</code>	Enables 64-Kbytes flash mode for the processor configurations <code>-v2</code> and <code>-v3</code> .
<code>--cpu</code>	Specifies the CPU model your application was compiled for.
<code>--disable_internal_eeprom</code>	Disables the internal EEPROM.
<code>--eeprom_size</code>	Specifies the size of the built-in EEPROM area.
<code>--enhanced_core</code>	Enables the enhanced instruction set.
<code>-p</code>	Specifies the device description file to be used.
<code>-v</code>	Specifies the processor configuration your application was compiled for.

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

- `--disable_interrupts` Disables the interrupt simulation.
- `--function_profiling` Analyzes your source code to find where the most time is spent during execution.

OPTIONS AVAILABLE FOR ALL C-SPY HARDWARE DEBUGGER DRIVERS

- `--drv_communication` Specifies the communication channel to be used between C-SPY and the target system.
- `--drv_communication_log` Logs the communication between C-SPY and the target system.
- `--drv_download_data` Enables downloading of constant data into RAM.
- `--drv_suppress_download` Suppresses download of the executable image.
- `--drv_verify_download` Verifies the executable image.

OPTIONS AVAILABLE FOR THE C-SPY POWER DEBUGGER DRIVER

- `--drv_power_debugger` Specifies the C-SPY Power Debugger driver to be used.

OPTIONS AVAILABLE FOR THE C-SPY JTAGICE MKII DRIVER, THE C-SPY DRAGON DRIVER, THE C-SPY JTAGICE3 DRIVER, AND THE C-SPY AVR ONE! DRIVER

- `--drv_set_exit_breakpoint` Sets a system breakpoint on the `exit` label.
- `--drv_set_getchar_breakpoint` Sets a system breakpoint on the `getchar` label.
- `--drv_set_putchar_breakpoint` Sets a system breakpoint on the `putchar` label.
- `--jtagice_do_hardware_reset` Makes the hardware reset every time the debugger is reset.

<code>--jtagice_leave_timers_running</code>	Ensures that the timers always run, even if the application is stopped.
<code>--jtagice_preserve_eeeprom</code>	Preserves the EEPROM contents even if the device is reprogrammed.
<code>--jtagice_restore_fuse</code>	Allows the debugger to modify the OCD enable fuse and preserve the EEPROM fuse at startup.

OPTIONS AVAILABLE FOR THE C-SPY JTAGICE MKII DRIVER, THE C-SPY DRAGON DRIVER, THE C-SPY ATMEL-ICE DRIVER, THE C-SPY POWER DEBUGGER DRIVER, THE C-SPY JTAGICE3 DRIVER, AND THE C-SPY AVR ONE! DRIVER

<code>--drv_preserve_app_section</code>	Preserves the application area of the flash memory during download.
<code>--drv_preserve_boot_section</code>	Preserves the boot area of the flash memory during download.

OPTIONS AVAILABLE FOR THE C-SPY JTAGICE MKII DRIVER AND THE C-SPY DRAGON DRIVER

<code>--jtagice_clock</code>	Specifies the speed of the JTAG clock.
------------------------------	--

OPTIONS AVAILABLE FOR THE C-SPY ATMEL-ICE DRIVER, THE C-SPY POWER DEBUGGER DRIVER, THE C-SPY JTAGICE3 DRIVER, AND THE C-SPY AVR ONE! DRIVER

<code>--avrone_jtag_clock</code>	Specifies the speed of the debugging interface.
<code>--drv_debug_port</code>	Specifies the debug interface.

OPTIONS AVAILABLE FOR THE C-SPY JTAGICE MKII DRIVER AND THE C-SPY DRAGON DRIVER

<code>--drv_use_PDI</code>	Makes the C-SPY driver communicate with the device using the PDI interface.
----------------------------	---

`--jtagicemkII_use_software_brea` Makes software breakpoints available.
`kpoints`

OPTIONS AVAILABLE FOR THE C-SPY DRAGON DRIVER

`--drv_dragon` Specifies the AVR Dragon driver to be used.

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.

--64bit_doubles

Syntax `--64bit_doubles`

For use with All C-SPY drivers.

Description Use this option to specify that 64-bit doubles are used instead of 32-bit doubles.



Project>Options>General Options>Target>Use 64-bit doubles

--64k_flash

Syntax `--64k_flash`

For use with All C-SPY drivers.

Description Use this option to enable 64-Kbytes flash mode for the processor configurations `-v2` and `-v3`.



Project>Options>General Options>Target>No RAMPZ register

--attach_to_running_target

Syntax `--attach_to_running_target`

For use with	<code>cspybat.</code>
	Note: This option might not be supported by the combination of C-SPY driver and device that you are using. If you are using this option with an unsupported combination, C-SPY produces a message.
Description	Use this option to make the debugger attach to a running application at its current location, without resetting the target system. If you have defined any breakpoints in your project, the C-SPY driver will set them during attachment. If the C-SPY driver cannot set them without stopping the target system, the breakpoints will be disabled. The option also suppresses download and the Run to option.



Project>Attach to Running Target

--avrone_jtag_clock

Syntax	<code>--avrone_jtag_clock=<i>speed</i></code>
Parameters	<i>speed</i> The JTAG or PDI clock frequency in Hz. Possible values are 0-65535000 Hz in steps of 1000 Hz.
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR ONE! driver The C-SPY JTAGICE3 driver
Description	Use this option to specify the speed of the debugging interface.



Project>Options>Debugger>Driver>Driver 1>Debug Port>Frequency in kHz

--backend

Syntax	<code>--backend {<i>driver options</i>}</code>
Parameters	<i>driver options</i> Any option available to the C-SPY driver you are using.

For use with	<code>cspybat</code> (mandatory).
Description	Use this option to send options to the C-SPY driver. All options that follow <code>--backend</code> will be passed to the C-SPY driver, and will not be processed by <code>cspybat</code> itself.



This option is not available in the IDE.

--code_coverage_file

Syntax	<code>--code_coverage_file file</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>file</i> The name of the destination file for the code coverage information.
For use with	<code>cspybat</code>
Description	Use this option to enable the generation of a text-based report file for code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file. Because most embedded applications do not terminate, you might have to use this option in combination with <code>--timeout</code> or <code>--cycles</code> . Note that this option requires that the C-SPY driver you are using supports code coverage. If you try to use this option with a C-SPY driver that does not support code coverage, an error message will be directed to <code>stderr</code> .
See also	<i>Code coverage</i> , page 207, <i>--cycles</i> , page 318, <i>--timeout</i> , page 333.



To set this option, choose **View>Code Coverage**, right-click and choose **Save As** when the C-SPY debugger is running.

--cpu

Syntax	<code>--cpu=cpu_name</code>
Parameters	<i>cpu_name</i> The CPU model, <code>xm128a1</code> , <code>m2560</code> , <code>tiny441</code> , etc.
For use with	All C-SPY drivers.

Description Use this option to specify the CPU model your application was compiled for. This option cannot be used together with `-v`.



Project>Options>General Options>Target>Processor configuration

--cycles

Syntax `--cycles cycles`

Note that this option must be placed before the `--backend` option on the command line.

Parameters *cycles*

The number of cycles to run.

For use with `cspybat`

Description Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted. Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.



This option is not available in the IDE.

--debugfile

Syntax `--debugfile filename`

Parameters *filename*

The name of the debug file to use.

For use with `cspybat`

This option can be placed both before and after the `--backend` option on the command line.

Description Use this option to make `cspybat` use the specified debug file instead of the one used in the generated `cspybat.bat` file.



This option is not available in the IDE.

--disable_internal_eeprom

Syntax `--disable_internal_eeprom`

For use with All C-SPY drivers.

Description Use this option to disable the internal EEPROM.



To set related options, choose:

Project>Options>General Options>Target>Utilize inbuilt EEPROM

--disable_interrupts

Syntax `--disable_interrupts`

For use with The C-SPY Simulator driver.

Description Use this option to disable the interrupt simulation.



To set this option, choose **Simulator>Interrupts** and deselect the **Enable simulation** option.

--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.



To set a related option, choose:

Project>Options>Debugger>Setup and deselect **Run to**.

--drv_communication

Syntax `--drv_communication=[COMn|USB]`

Parameters

`COMn` A serial communication port. *n* can be between 1 and 32. Note that `COMn` is not used in AVR ONE! and JTAGICE3.

	USB	The USB port. Can only be used by the AVR ONE!, JTAGICE mkII, JTAGICE3, and AVR Dragon drivers.
For use with		All C-SPY hardware drivers.
Description		Use this option to specify the communication channel to be used between C-SPY and the target system.



Project>Options>Debugger>Driver

--drv_communication_log

Syntax	<code>--drv_communication_log=<i>filename</i></code>	
Parameters	<i>filename</i>	The name of the log file.
For use with		All C-SPY hardware drivers.
Description		Use this option to log the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the communication protocol is required.



Project>Options>Debugger>Atmel-ICE>Communication>Log communication

Project>Options>Debugger>Power Debugger>Communication>Log communication

Project>Options>Debugger>AVR ONE!>Communication>Log communication





Project>Options>Debugger>JTAGICE3>Communication>Log communication

Project>Options>Debugger>JTAGICE mkII>Serial Port>Log communication


Project>Options>Debugger>Dragon>Communication>Log communication

--drv_debug_port

Syntax	<code>--drv_debug_port={autodetect debugwire pdi jtag tpi updi}</code>	
Parameters	<code>autodetect</code>	Specifies auto-detection of the debug interface.
	<code>debugwire</code>	Specifies the debugWIRE debug interface.

	<code>pdi</code>	Specifies the PDI debug interface.
	<code>jtag</code>	Specifies the JTAG debug interface.
	<code>tpi</code>	Specifies the TPI programming interface.
	<code>updi</code>	Specifies the UPDI debug interface.
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR ONE! driver The C-SPY JTAGICE3 driver	
Description	Use this option to specify the debug interface.  Project>Options>Debugger>Atmel-ICE>Atmel-ICE 1>Debug Port  Project>Options>Debugger>Power Debugger>Power Debugger 1>Debug Port  Project>Options>Debugger>AVR ONE!>AVR ONE! 1>Debug Port  Project>Options>Debugger>JTAGICE3>JTAGICE3 1>Debug Port	

--drv_download_data

Syntax	<code>--drv_download_data</code>
For use with	All C-SPY hardware drivers.
Description	Use this option to enable downloading of constant data into RAM.  Project>Options>Debugger>Driver>Driver 1>Allow download to RAM

--drv_dragon

Syntax	<code>--drv_dragon</code>
For use with	The C-SPY AVR Dragon driver.
Description	Use this option to specify the AVR Dragon driver to be used.

**Project>Options>Debugger>Driver****--drv_power_debugger**

Syntax	--drv_power_debugger
For use with	The C-SPY Power Debugger driver.
Description	Use this option to specify the C-SPY Power Debugger driver to be used.

**Project>Options>Debugger>Driver****--drv_preserve_app_section**

Syntax	--drv_preserve_app_section
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR ONE! driver The C-SPY AVR Dragon driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option to preserve the application area of the flash memory during download.

**Project>Options>Debugger>Driver>Driver 2>Preserve FLASH>Application Area****--drv_preserve_boot_section**

Syntax	--drv_preserve_boot_section
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR ONE! driver

	The C-SPY JTAGICE3 driver
	The C-SPY AVR Dragon driver
	The C-SPY JTAGICE mkII driver
Description	Use this option to preserve the boot area of the flash memory during download.



Project>Options>Debugger>Driver>Driver 2>Preserve FLASH>Boot Area

--drv_set_exit_breakpoint

Syntax	--drv_set_exit_breakpoint
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR ONE! driver The C-SPY AVR Dragon driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option in the CLIB runtime environment to set a system breakpoint on the <code>exit</code> label. This consumes a hardware breakpoint.
See also	<i>Breakpoint consumers</i> , page 110



Project>Options>Debugger>Driver>Driver 2>System breakpoints on>exit

--drv_set_getchar_breakpoint

Syntax	--drv_set_getchar_breakpoint
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE mkII driver

Description	The C-SPY JTAGICE3 driver Use this option in the CLIB runtime environment to set a system breakpoint on the <code>getchar</code> label. This consumes a hardware breakpoint.
See also	<i>Breakpoint consumers</i> , page 110



Project>Options>Debugger>Driver>Driver 2>System breakpoints on>getchar

--drv_set_putchar_breakpoint

Syntax	<code>--drv_set_putchar_breakpoint</code>
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option in the CLIB runtime environment to set a system breakpoint on the <code>putchar</code> label. This consumes a hardware breakpoint.
See also	<i>Breakpoint consumers</i> , page 110



Project>Options>Debugger>Driver>Driver 2>System breakpoints on>putchar

--drv_suppress_download

Syntax	<code>--drv_suppress_download</code>
For use with	All C-SPY hardware drivers.
Description	Use this option to disable the downloading of code, preserving the current contents of the flash memory.



Project>Options>Debugger>Driver>Driver 1>Suppress download

--drv_use_PDI

Syntax	--drv_use_PDI
For use with	The C-SPY AVR Dragon driver The C-SPY JTAGICE mkII driver
Description	Use this option if you want the C-SPY driver to communicate with the device using the PDI interface.



Project>Options>Debugger>Driver>Driver 1>Use PDI

--drv_verify_download

Syntax	--drv_verify_download
For use with	All C-SPY hardware drivers.
Description	Use this option to verify that the downloaded code image can be read back from target memory with the correct contents.



Project>Options>Debugger>Driver>Driver 1>Target Consistency Check>Verify All

--eeprom_size

Syntax	--eeprom_size= <i>size</i>
Parameters	<i>size</i> The size of the built-in EEPROM area in bytes.
For use with	All C-SPY drivers.
Description	Use this option to specify the size of the built-in EEPROM area. Do not use together with --cpu.



Project>Options>General Options>Target>Utilize inbuilt EEPROM

--enhanced_core

Syntax	<code>--enhanced_core</code>
For use with	All C-SPY drivers.
Description	Use this option to enable the enhanced instruction set; the instructions <code>MOVW</code> , <code>MUL</code> , <code>MULS</code> , <code>MULSU</code> , <code>FMUL</code> , <code>FMULS</code> , <code>FMULSU</code> , <code>LPM Rd, Z</code> , <code>LPM Rd, Z+</code> , <code>ELPM Rd, Z</code> , <code>ELPM Rd, Z+</code> , and <code>SPM</code> .



Project>Options>General Options>Target>Enhanced core

-f

Syntax	<code>-f filename</code>
Parameters	<i>filename</i> A text file that contains the command line options (default filename extension <code>xcl</code>).
For use with	<code>cspybat</code> This option can be placed either before or after the <code>--backend</code> option on the command line.
Description	Use this option to make <code>cspybat</code> read command line options from the 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.



To set this option, use **Project>Options>Debugger>Extra Options**.

--function_profiling

Syntax	<code>--function_profiling filename</code>
Parameters	<i>filename</i> The name of the log file where the profiling data is saved.

For use with	The C-SPY simulator driver.
Description	Use this option to find the functions in your source code where the most time is spent during execution. The profiling information is saved to the specified file. For more information about function profiling, see <i>Profiling</i> , page 197.



***C-SPY driver*>Function Profiling**

--jtagice_clock

Syntax	<code>--jtagice_clock=<i>speed</i></code>
Parameters	<i>speed</i> The JTAG clock frequency in Hz. Possible values are 0-3570000 Hz.
For use with	The C-SPY AVR Dragon driver The C-SPY JTAGICE mkII driver
Description	Use this option to specify the speed of the JTAG clock.



Project>Options>Debugger>Driver>Driver 1>JTAG Port>Frequency in Hz


--jtagice_do_hardware_reset

Syntax	<code>--jtagice_do_hardware_reset</code>
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option to make the hardware reset every time the debugger is reset.




Project>Options>Debugger>Driver>Driver 2>Hardware reset on C-SPY reset

--jtagice_leave_timers_running

Syntax	--jtagice_leave_timers_running
For use with	The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE mkII driver
Description	Use this option to ensure that the timers always run, even if the application is stopped.  Project>Options>Debugger>Driver>Driver 2>Run timers in stopped mode

--jtagice_preserve_eeprom

Syntax	--jtagice_preserve_eeprom
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver
Description	Use this option to preserve the EEPROM contents even if device is reprogrammed.  Project>Options>Debugger>Driver>Driver 2>Preserve EEPROM contents even if device is reprogrammed

--jtagice_restore_fuse

Syntax	--jtagice_restore_fuse
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver

	The C-SPY JTAGICE mkII driver
	The C-SPY JTAGICE3 driver
Description	Use this option to allow the debugger to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch decreases the life span of the chip.



Project>Options>Debugger>Driver>Driver 2>Restore fuses when ending debug session

--jtagicemkII_use_software_breakpoints

Syntax	--jtagicemkII_use_software_breakpoints
For use with	The C-SPY Atmel-ICE driver The C-SPY Power Debugger driver The C-SPY AVR Dragon driver The C-SPY AVR ONE! driver The C-SPY JTAGICE mkII driver The C-SPY JTAGICE3 driver

Description	Use this option to make software breakpoints available.
-------------	---



Project>Options>Debugger>Driver>Driver 2>Enable software breakpoints

--leave_target_running

Syntax	--leave_target_running
For use with	cspybat.

Any C-SPY hardware debugger driver.

Note: Even if this option is supported by the C-SPY driver you are using, there might be device-specific limitations.

Description	Use this option to make the debugger leave the application running on the target hardware after the debug session is closed.
-------------	--



Any existing breakpoints will not be automatically removed. You might want to consider disabling all breakpoints before using this option.



C-SPY driver>Leave Target Running

--macro

Syntax	<code>--macro filename</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The C-SPY macro file to be used (filename extension <code>mac</code>).
For use with	<code>cspybat</code>
Description	Use this option to specify a C-SPY macro file to be loaded before executing the target application. This option can be used more than once on the command line.
See also	<i>Briefly about using C-SPY macros</i> , page 252.



Project>Options>Debugger>Setup>Setup macros>Use macro file

--macro_param

Syntax	<code>--macro_param [param=value]</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>param = value</i> <i>param</i> is a parameter defined using the <code>__param</code> C-SPY macro construction. <i>value</i> is a value.
For use with	<code>cspybat</code>
Description	Use this option to assign a value to a C-SPY macro parameter. This option can be used more than once on the command line.
See also	<i>Macro parameters</i> , page 259.

**Project>Options>Debugger>Extra Options****-p**

Syntax	<code>-p filename</code>
Parameters	<i>filename</i> The device description file to be used.
For use with	All C-SPY drivers.
Description	Use this option to specify the device description file to be used.
See also	<i>Selecting a device description file</i> , page 51.

**Project>Options>Debugger>Setup>Device description file****--plugin**

Syntax	<code>--plugin filename</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The plugin file to be used (filename extension <code>dll</code>).
For use with	<code>cspybat</code>
Description	Certain C/C++ standard library functions, for example <code>printf</code> , can be supported by C-SPY—for example, the C-SPY Terminal I/O window—instead of by real hardware devices. To enable such support in <code>cspybat</code> , a dedicated plugin module called <code>avrllibsupportbat.dll</code> located in the <code>avr\bin</code> directory must be used. Use this option to include this plugin during the debug session. This option can be used more than once on the command line. Note: You can use this option to include also other plugin modules, but in that case the module must be able to work with <code>cspybat</code> specifically. This means that the C-SPY plugin modules located in the <code>common\plugin</code> directory cannot normally be used with <code>cspybat</code> .



Project>Options>Debugger>Plugins

--program_fuses_after_download

Syntax	<code>--program_fuses_after_download fuse=value[,fuse=value,...]</code>
Parameters	<p><i>fuse</i> The fuse or lock bit to set a value for. Choose between:</p> <ul style="list-style-type: none"> <code>--lock_bits_value</code> <code>--low_fuse_value</code> <code>--high_fuse_value</code> <code>--extended_fuse_value</code> <code>--fuse_byte0_value</code> <code>--fuse_byte1_value</code> <code>--fuse_byte2_value</code> <code>--fuse_byte4_value</code> <code>--fuse_byte5_value</code> <p><i>value</i> 0x00-0xFF</p>
For use with	<p>The C-SPY Atmel-ICE driver</p> <p>The C-SPY Power Debugger driver</p> <p>The C-SPY AVR ONE! driver</p> <p>The C-SPY Dragon driver</p> <p>The C-SPY JTAGICE mkII driver</p> <p>The C-SPY JTAGICE3 driver.</p>
Description	Use this option to write values to device-specific on-chip fuses and lock bits after downloading your application to the device.
Example	<code>--program_fuses_after_download --fuse_byte3_value=0x20</code>
See also	For more information about fuse settings, see the device-specific documentation provided by Atmel® Corporation.



To set related options, choose ***C-SPY driver*>Fuse Handler**.

--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.

--timeout

Syntax `--timeout milliseconds`

Note that this option must be placed before the `--backend` option on the command line.

Parameters *milliseconds*

The number of milliseconds before the execution stops.

For use with `cspybat`

Description Use this option to limit the maximum allowed execution time.



This option is not available in the IDE.

-v

Syntax `-v {0|1|2|3|4|5|6}`

Parameters

- 0 A maximum of 256 bytes data and 8 Kbytes code. Default memory model: Tiny.
- 1 A maximum of 64 Kbytes data and 8 Kbytes code. Default memory model: Tiny.
- 2 A maximum of 256 bytes data and 128 Kbytes code. Default memory model: Tiny.
- 3 A maximum of 64 Kbytes data and 128 Kbytes code. Default memory model: Tiny.
- 4 A maximum of 16 Mbytes data and 128 Kbytes code. Default memory model: Small.

- 5 A maximum of 64 Kbytes data and 8 Mbytes code. Default memory model: Small.
- 6 A maximum of 16 Mbytes data and 8 Mbytes code. Default memory model: Small.

For use with

All C-SPY drivers.

Description

Use this option to specify the processor configuration your application was compiled for. This option cannot be used together with `--cpu`.

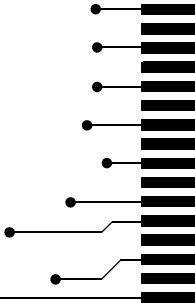


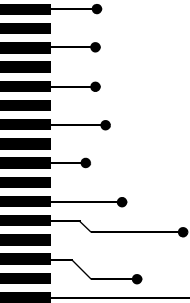
Project>Options>General Options>Target>Processor configuration

Part 4. Additional reference information

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

- Debugger options
- Additional information on C-SPY drivers





Debugger options

- Setting debugger options
- Reference information on general debugger options
- Reference information on C-SPY hardware debugger driver options

Setting debugger options

Before you start the C-SPY debugger you might need to set some options—both C-SPY generic options and options required for the target system (C-SPY driver-specific options).

To set debugger options in the IDE:

- 1 Choose **Project>Options** to display the **Options** dialog box.
- 2 Select **Debugger** in the **Category** list.

For more information about the generic options, see *Reference information on general debugger options*, page 338.

- 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 AVR ONE! driver	<i>AVR ONE! 1</i> , page 348 <i>AVR ONE! 2</i> , page 350 <i>Communication</i> , page 346 <i>Extra Options</i> , page 347
C-SPY Atmel-ICE driver	<i>Atmel-ICE 1</i> , page 342 <i>Atmel-ICE 2</i> , page 345 <i>Communication</i> , page 346 <i>Extra Options</i> , page 347

Table 35: Options specific to the C-SPY drivers you are using

C-SPY driver	Available options pages
C-SPY JTAGICE3 driver	<i>JTAGICE3 1</i> , page 351 <i>JTAGICE3 2</i> , page 353 <i>Communication</i> , page 346 <i>Extra Options</i> , page 347
C-SPY JTAGICE mk II driver	<i>JTAGICE mkII 1</i> , page 354 <i>JTAGICE mkII 2</i> , page 357 <i>Serial Port</i> , page 358 <i>Extra Options</i> , page 347
C-SPY Dragon driver	<i>Dragon 1</i> , page 359 <i>Dragon 2</i> , page 361 <i>Communication</i> , page 346 <i>Extra Options</i> , page 347
C-SPY Power Debugger driver	<i>Power Debugger 1</i> , page 362 <i>Power Debugger 2</i> , page 364 <i>Communication</i> , page 346 <i>Extra Options</i> , page 347
C-SPY Simulator	<i>Extra Options</i> , page 347
Third-party driver	<i>Third-Party Driver options</i> , page 365 <i>Extra Options</i> , page 347

Table 35: Options specific to the C-SPY drivers you are using (Continued)

- 5** To restore all settings to the default factory settings, click the **Factory Settings** button.
- 6** When you have set all the required options, click **OK** in the **Options** dialog box.

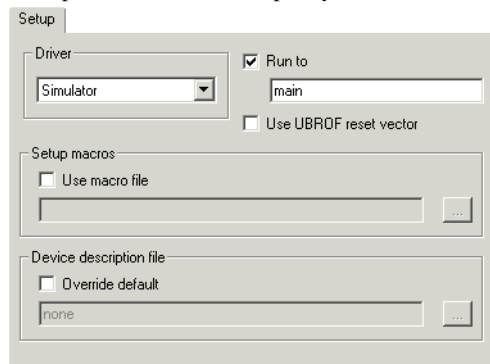
Reference information on general debugger options

Reference information about:

- Setup
- Images
- Plugins

Setup

The general **Setup** options select the C-SPY driver, the setup macro file, and device description file to use, and specify which default source code location to run to.



Driver

Selects the C-SPY driver for the target system you have.

Run to

Specifies the location C-SPY runs to when the debugger starts after a reset. By default, C-SPY runs to the `main` function.

To override the default location, specify the name of a different location you want C-SPY to run to. You can specify assembler labels or whatever can be evaluated as such, for example function names.

If the option is deselected, the program counter will contain the regular hardware reset address at each reset.

See also *Executing from reset*, page 50.

Use UBROF reset vector

Makes the debugger use the reset vector specified as the entry label `__program_start`, see the *IDE Project Management and Building Guide for AVR*. By default, the reset vector is set to `0x0`.

Setup macros

Registers the contents of a setup macro file in the C-SPY startup sequence. Select **Use macro file** and specify the path and name of the setup file, for example `SetupSimple.mac`. If no extension is specified, the extension `mac` is assumed. A browse button is available for your convenience.

Device description file

A default device description file is selected automatically based on your project settings. To override the default file, select **Override default** and specify an alternative file. A browse button is available for your convenience.

For information about the device description file, see *Modifying a device description file*, page 55.

IAR-specific device description files for each AVR device are provided in the directory `avr\config` and have the filename extension `.ddf`.

Images

The **Images** options control the use of additional debug files to be downloaded.

The screenshot shows a dialog box titled "Images" with a light green background. It contains three vertically stacked sections, each for configuring an additional debug image. Each section starts with a checkbox labeled "Download extra image". Below this is a "Path:" label followed by a text input field and a browse button (three dots in a box). To the right of the path field is an "Offset:" label followed by a text input field. Further right is another checkbox labeled "Debug info only".

Download extra Images

Controls the use of additional debug files to be downloaded:

Path

Specify the debug file to be downloaded. A browse button is available for your convenience.

Offset

Specify an integer that determines the destination address for the downloaded debug file.

Debug info only

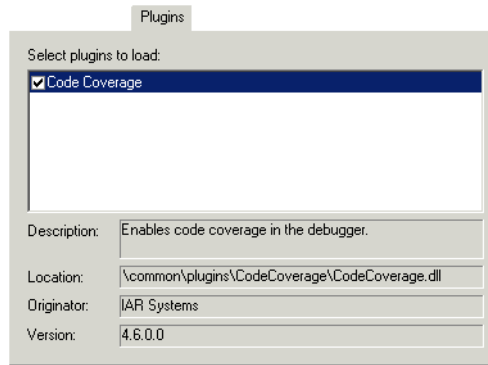
Makes the debugger download only debug information, and not the complete debug file.

If you want to download more than three images, use the related C-SPY macro, see `__loadImage`, page 275.

For more information, see *Loading multiple images*, page 53.

Plugins

The **Plugins** options select the C-SPY plugin modules to be loaded and made available during debug sessions.



Select plugins to load

Selects the plugin modules to be loaded and made available during debug sessions. The list contains the plugin modules delivered with the product installation.

Description

Describes the plugin module.

Location

Informs about the location of the plugin module.

Generic plugin modules are stored in the `common\plugins` directory. Target-specific plugin modules are stored in the `avr\plugins` directory.

Originator

Informs about the originator of the plugin module, which can be modules provided by IAR Systems or by third-party vendors.

Version

Informs about the version number.

Reference information on C-SPY hardware debugger driver options

Reference information about:

- *AVR ONE! 1*, page 348
- *AVR ONE! 2*, page 350
- *Communication*, page 346
- *Extra Options*, page 347
- *Serial Port*, page 358
- *Atmel-ICE 1*, page 342
- *Atmel-ICE 2*, page 345
- *JTAGICE3 1*, page 351
- *JTAGICE3 2*, page 353
- *JTAGICE mkII 1*, page 354
- *JTAGICE mkII 2*, page 357
- *Dragon 1*, page 359
- *Dragon 2*, page 361
- *Power Debugger 1*, page 362
- *Power Debugger 2*, page 364
- *Third-Party Driver options*, page 365

Atmel-ICE 1

The **Atmel-ICE 1** options control the C-SPY driver for Atmel-ICE.

Atmel-ICE 1

Debug Port

Auto detect Frequency in kHz: 100 KHz

JTAG 100

PDI

JTAG Port

Target device is part of a JTAG daisy chain

Devices: Instruction bits:

Before:

After:

Download control

Suppress Download

Allow download to RAM

Target Consistency Check

None

Verify Boundaries

Verify All

Debug Port

Selects the communication type. Choose between:

Auto detect	Auto-detects JTAG or PDI. The JTAG Port options are not available with this setting. A JTAG device must be first in a JTAG chain.
JTAG	Specifies JTAG only mode.
PDI	Specifies PDI only mode.
Frequency in Hz	<p>Defines the debug port clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in kHz and can be any value from 1 to 65536. The default value is 100 kHz.</p> <p>A too small value (less than 28 kHz) might cause the communication to time out.</p> <p>A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.</p>

JTAG Port

Configures the JTAG port.

Target device is part of a JTAG daisy chain	If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.
Devices Before	Specify the number of JTAG data bits before the device in the JTAG chain.
Devices After	Specify the number of JTAG data bits after the device in the JTAG chain.
Instruction bits Before	Specify the number of JTAG instruction register bits before the device in the JTAG chain.
Instruction bits After	Specify the number of JTAG instruction register bits after the device in the JTAG chain.

Download control

Controls the download.

Suppress download

Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify all** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Allow download to RAM

Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.

Target consistency check

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:

None, target consistency check is not performed.

Verify Boundaries, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.

Verify All, checks every byte after loading. This is a slow, but complete, check of the memory.

Atmel-ICE 2

The **Atmel-ICE 2** options control the C-SPY driver for Atmel-ICE.

Atmel-ICE 2

Run timers in stopped mode

Preserve EEPROM contents even if device is reprogrammed

Restore fuses when ending debug session

Enable software breakpoints

System breakpoints on

exit

putchar

getchar

Preserve FLASH

None

Boot Area

Application Area

Run timers in stopped mode

Runs the timers even if the program is stopped.

Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. If this option is deselected, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

Restore fuses when ending debug session

Enables C-SPY to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch will decrease the life span of the chip.

Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 105.

System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY **Terminal I/O** window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options **exit**, **putchar**, and **getchar**, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

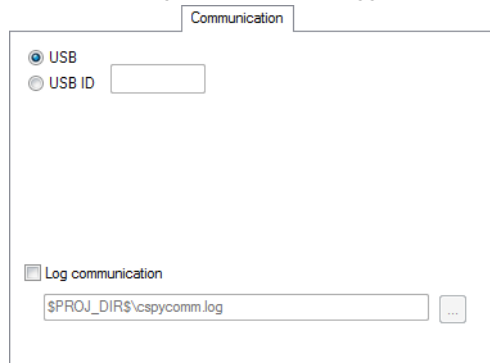
For more information, see *Breakpoint consumers*, page 110.

Preserve FLASH

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

Communication

The **Communication** options control the C-SPY driver for Atmel-ICE, AVR ONE!, JTAGICE3, Dragon, or Power Debugger.



USB

Specifies single emulator mode. Use this option for the USB port and if you have one device connected to your host computer.

USB ID

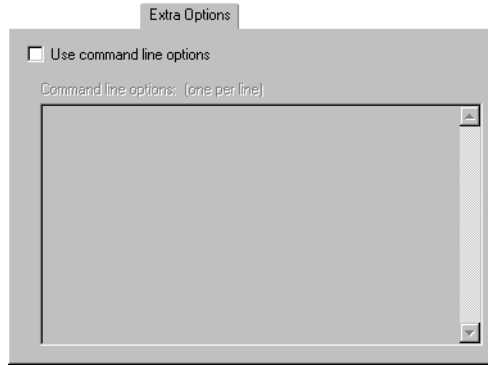
Specifies multi-emulator mode. Use this option for the USB port and if you have more than one device connected to your host computer. Specify the serial number of the device that you want to connect to, or the USB ID visible in the **Log Messages** window. The serial number, for example **ONE00737**, is printed on the tag underneath the 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.

Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.



Use command line options

Specify command line arguments that are not supported by the IDE to be passed to C-SPY.

Note that it is possible to use the `/args` option to pass command line arguments to the debugged application.

Syntax: `/args arg0 arg1 ...`

Multiple lines with `/args` are allowed, for example:

```
/args --logfile log.txt
```

```
/args --verbose
```

If you use `/args`, these variables must be defined in your application:

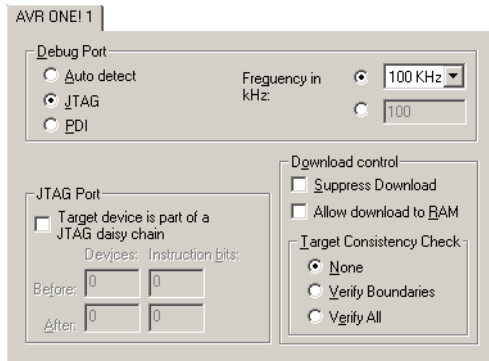
```
/* __argc, the number of arguments in __argv. */
__no_init int __argc;
```

```
/* __argv, an array of pointers to strings that holds the
arguments; must be large enough to fit the number of
parameters.*/
__no_init const char * __argv[MAX_ARGS];
```

```
/* __argvbuf, a storage area for __argv; must be large enough to
hold all command line parameters. */
__no_init __root char __argvbuf[MAX_ARG_SIZE];
```

AVR ONE! I

The **AVR ONE! 1** options control the C-SPY driver for AVR ONE!.



Debug Port

Selects the communication type. Choose between

- | | |
|------------------------|---|
| Auto detect | Auto-detects JTAG or PDI. The JTAG Port options are not available in this mode. A JTAG device must be first in a JTAG chain. |
| JTAG | Specifies JTAG only mode. |
| PDI | Specifies PDI only mode. |
| Frequency in Hz | Defines the debug port clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in kHz and can be any value from 1 to 65536. The default value is 100 kHz.

A too small value (less than 28 kHz) might cause the communication to time out.

A too large value will result in an unexpected error while debugging, such as an execution/read/write failure. |

JTAG Port

Configures the JTAG port.

- | | |
|--|---|
| Target device is part of a JTAG daisy chain | If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option. |
|--|---|

Devices Before	Specify the number of JTAG data bits before the device in the JTAG chain.
Devices After	Specify the number of JTAG data bits after the device in the JTAG chain.
Instruction bits Before	Specify the number of JTAG instruction register bits before the device in the JTAG chain.
Instruction bits After	Specify the number of JTAG instruction register bits after the device in the JTAG chain.

Download control

Controls the download.

Suppress download	<p>Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.</p> <p>If this option is combined with the Verify all option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.</p>
Allow download to RAM	Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.
Target consistency check	<p>Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:</p> <p>None, target consistency check is not performed.</p> <p>Verify Boundaries, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.</p> <p>Verify All, checks every byte after loading. This is a slow, but complete, check of the memory.</p>

AVR ONE! 2

The **AVR ONE! 2** options control the C-SPY driver for AVR ONE!.

Run timers in stopped mode

Runs the timers even if the program is stopped.

Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

Hardware reset on C-SPY reset

Causes a reset in the debugger to also do a hardware reset by pulling down the nSRST signal.

Restore fuses when ending debug session

Enables C-SPY to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch will decrease the life span of the chip.

Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 105.

System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options **exit**, **putchar**, and **getchar**, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

For more information, see *Breakpoint consumers*, page 110.

Preserve FLASH

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

JTAGICE3 I

The **JTAGICE3 1** options control the C-SPY driver for JTAGICE3.

Debug Port

Selects the communication type. Choose between

- | | |
|--------------------|---|
| Auto detect | Auto-detects JTAG or PDI. The JTAG Port options are not available with this setting. A JTAG device must be first in a JTAG chain. |
| JTAG | Specifies JTAG only mode. |
| PDI | Specifies PDI only mode. |

Frequency in Hz	Defines the debug port clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in kHz and can be any value from 1 to 65536. The default value is 100 kHz.
	A too small value (less than 28 kHz) might cause the communication to time out.
	A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.

JTAG Port

Configures the JTAG port.

Target device is part of a JTAG daisy chain	If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.
Devices Before	Specify the number of JTAG data bits before the device in the JTAG chain.
Devices After	Specify the number of JTAG data bits after the device in the JTAG chain.
Instruction bits Before	Specify the number of JTAG instruction register bits before the device in the JTAG chain.
Instruction bits After	Specify the number of JTAG instruction register bits after the device in the JTAG chain.

Download control

Controls the download.

Suppress download	Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.
	If this option is combined with the Verify all option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Allow download to RAM

Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.

Target consistency check

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:

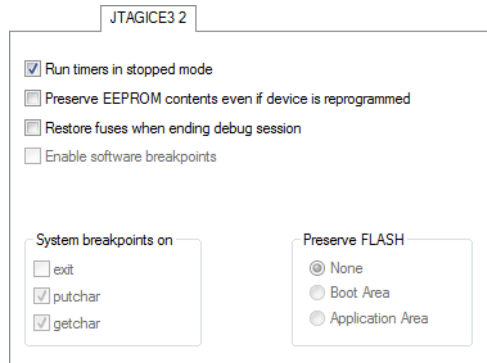
None, target consistency check is not performed.

Verify Boundaries, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.

Verify All, checks every byte after loading. This is a slow, but complete, check of the memory.

JTAGICE3 2

The **JTAGICE3 2** options control the C-SPY driver for JTAGICE3.



JTAGICE3 2

Run timers in stopped mode

Preserve EEPROM contents even if device is reprogrammed

Restore fuses when ending debug session

Enable software breakpoints

System breakpoints on

exit

putchar

getchar

Preserve FLASH

None

Boot Area

Application Area

Run timers in stopped mode

Runs the timers even if the program is stopped.

Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

Restore fuses when ending debug session

Enables C-SPY to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch will decrease the life span of the chip.

Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 105.

System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options `exit`, `putchar`, and `getchar`, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

For more information, see *Breakpoint consumers*, page 110.

Preserve FLASH

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

JTAGICE mkII I

The **JTAGICE mkII 1** options control the C-SPY drivers for JTAGICE mkII and Dragon.

The screenshot shows the 'JTAGICE mkII 1' options dialog. It has a title bar with the text 'JTAGICE mkII 1'. The dialog is divided into several sections:

- Use PDI:** A checkbox that is currently unchecked.
- JTAG Port:**
 - Frequency in Hz: A dropdown menu set to '100 KHz' and a text input field containing '100000'.
 - Target device is part of a JTAG daisy chain: An unchecked checkbox.
 - Devices: Instruction bits: A section with two columns of input fields. The 'Before' column has two fields, both containing '0'. The 'After' column has two fields, both containing '0'.
- Communication:** A section with three radio buttons: 'USB', 'USB ID' (with an adjacent text input field), and 'Serial port' (which is selected).
- Download control:** A section with two unchecked checkboxes: 'Suppress Download' and 'Allow download to RAM'.
- Target Consistency Check:** A section with three radio buttons: 'None' (selected), 'Verify Boundaries', and 'Verify All'.

Use PDI

Enables communication with the device using the PDI interface.

Communication

Selects the communication type. Choose between:

USB	Selects the USB port. Use this setting if you have one AVR JTAGICE mkII device connected to your host computer.
USB ID	Selects the USB port. Use this setting if you have more than one AVR JTAGICE mkII device connected to your host computer. Specify the serial number of the device that you want to connect to. The serial number is printed on the tag underneath the device.
Serial port	Selects the serial port. To configure the serial port, select the Serial Port page in the Options dialog box and then choose a port from the Default communication drop-down list. By default, the COM1 port is used at 38400 baud.

JTAG Port

Configures the JTAG port.

Frequency in Hz	<p>Defines the JTAG clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in Hz and can be any value from 5000 to 1000000. The frequency value is rounded down to nearest available in the JTAGICE. The default value is 100 kHz.</p> <p>A too small value (less than 28000) might cause the communication to time out. The value must not be greater than 1/4 of the target CPU clock frequency.</p> <p>A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.</p>
Target device is part of a JTAG daisy chain	If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.
Devices Before	Specify the number of JTAG data bits before the device in the JTAG chain.
Devices After	Specify the number of JTAG data bits after the device in the JTAG chain.

Instruction bits Before Specify the number of JTAG instruction register bits before the device in the JTAG chain.

Instruction bits After Specify the number of JTAG instruction register bits after the device in the JTAG chain.

Download control

Controls the download.

Suppress download Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.

If this option is combined with the **Verify all** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Allow download to RAM Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.

Target consistency check Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:

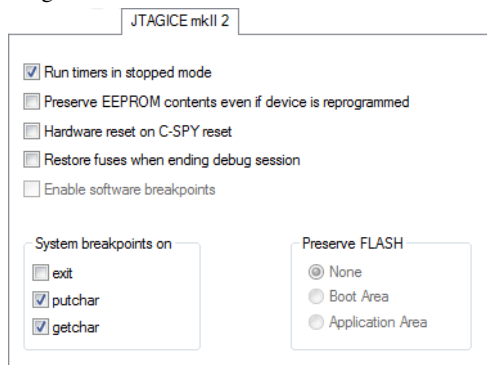
None, target consistency check is not performed.

Verify Boundaries, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.

Verify All, checks every byte after loading. This is a slow, but complete, check of the memory.

JTAGICE mkII 2

The **JTAGICE mkII 2** options control the C-SPY drivers for JTAGICE mkII and Dragon.



JTAGICE mkII 2

Run timers in stopped mode

Preserve EEPROM contents even if device is reprogrammed

Hardware reset on C-SPY reset

Restore fuses when ending debug session

Enable software breakpoints

System breakpoints on

exit

putchar

getchar

Preserve FLASH

None

Boot Area

Application Area

Run timers in stopped mode

Runs the timers even if the program is stopped.

Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

Hardware reset on C-SPY reset

Causes a reset in the debugger to also do a hardware reset by pulling down the nSRST signal.

Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 105.

System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options `exit`, `putchar`, and `getchar`, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

For more information, see *Breakpoint consumers*, page 110.

Preserve FLASH

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

Serial Port

The Serial Port options determine how the serial port should be used.

The screenshot shows a dialog box titled "Serial Port". It contains two main sections. The first section, "Default communication", has a checked checkbox and several dropdown menus: "Port" (set to COM 1), "Baud" (set to 9600), "Parity" (set to None), "Data bits" (set to 8 data bits), "Stop bits" (set to 1 stop bit), and "Handshaking" (set to None). The second section, "Log communication", also has a checked checkbox and a text input field containing "cspycomm.log" with a browse button to its right.

Default communication

Sets the default communication to use the port you specify and use it at 38400 baud.

Port

Selects one of the supported ports: **COM1** (default), **COM2**, ..., **COM32**.

Baud

Selects the baud rate.

If the debug session is terminated unexpectedly (by a fatal error, for instance), you might have to switch the emulator on and off to make it reconnect—first at default rate and then at the selected rate.

Parity

Selects the parity; only **None** is allowed.

Data bits

Selects the number of data bits; only **8 data bits** is allowed.

Stop bits

Selects the number of stop bits: **1 stop bit** or **2 stop bits**.

Handshaking

Selects the handshaking method, **None** or **RTSCTS**.

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.

Dragon I

The **Dragon 1** options control the C-SPY drivers for Dragon.

The screenshot shows the 'Dragon 1' options dialog box. It contains the following settings:

- Use PDI
- JTAG Port**
 - Frequency in Hz: 100 KHz (dropdown menu) 100000
 - Target device is part of a JTAG daisy chain

Devices:	Instruction bits:
Before: <input type="text" value="0"/>	<input type="text" value="0"/>
After: <input type="text" value="0"/>	<input type="text" value="0"/>
- Download control**
 - Suppress Download
 - Allow download to RAM
- Target Consistency Check**
 - None
 - Verify Boundaries
 - Verify All

Use PDI

Enables communication with the device using the PDI interface.

JTAG Port

Configures the JTAG port.

Frequency in Hz	<p>Defines the JTAG clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in Hz and can be any value from 5000 to 1000000. The frequency value is rounded down to nearest available in the JTAGICE. The default value is 100 kHz.</p> <p>A too small value (less than 28000) might cause the communication to time out. The value must not be greater than 1/4 of the target CPU clock frequency.</p> <p>A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.</p>
Target device is part of a JTAG daisy chain	<p>If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.</p>
Devices Before	<p>Specify the number of JTAG data bits before the device in the JTAG chain.</p>
Devices After	<p>Specify the number of JTAG data bits after the device in the JTAG chain.</p>
Instruction bits Before	<p>Specify the number of JTAG instruction register bits before the device in the JTAG chain.</p>
Instruction bits After	<p>Specify the number of JTAG instruction register bits after the device in the JTAG chain.</p>

Download control

Controls the download.

Suppress download	<p>Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. The implicit RESET performed by C-SPY at startup is not disabled, though.</p> <p>If this option is combined with the Verify all option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.</p>
--------------------------	--

Allow download to RAM

Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.

Target consistency check

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:

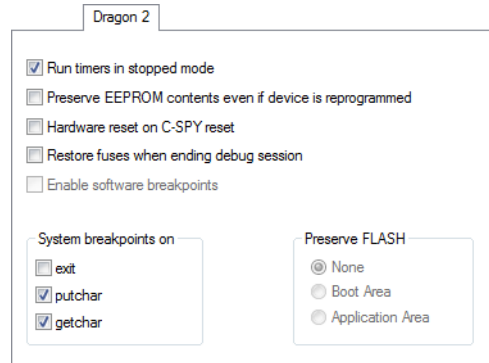
None, target consistency check is not performed.

Verify Boundaries, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.

Verify All, checks every byte after loading. This is a slow, but complete, check of the memory.

Dragon 2

The **Dragon 2** options control the C-SPY drivers for Dragon.



Dragon 2

- Run timers in stopped mode
- Preserve EEPROM contents even if device is reprogrammed
- Hardware reset on C-SPY reset
- Restore fuses when ending debug session
- Enable software breakpoints

System breakpoints on

- exit
- putchar
- getchar

Preserve FLASH

- None
- Boot Area
- Application Area

Run timers in stopped mode

Runs the timers even if the program is stopped.

Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. When this option is not used, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

Hardware reset on C-SPY reset

Causes a reset in the debugger to also do a hardware reset by pulling down the nSRST signal.

Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 105.

System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY Terminal I/O window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options `exit`, `putchar`, and `getchar`, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

For more information, see *Breakpoint consumers*, page 110.

Power Debugger 1

The **Power Debugger 1** options control the C-SPY driver for Power Debugger.

Debug Port

Selects the communication type. Choose between:

Auto detect

Auto-detects JTAG or PDI. The JTAG Port options are not available with this setting. A JTAG device must be first in a JTAG chain.

JTAG	Specifies JTAG only mode.
PDI	Specifies PDI only mode.
Frequency in Hz	<p>Defines the debug port clock speed. You can choose the value from the drop-down list, or enter a custom value in the text box. The value is in kHz and can be any value from 1 to 65536. The default value is 100 kHz.</p> <p>A too small value (less than 28 kHz) might cause the communication to time out.</p> <p>A too large value will result in an unexpected error while debugging, such as an execution/read/write failure.</p>

JTAG Port

Configures the JTAG port.

Target device is part of a JTAG daisy chain	If the AVR CPU is not alone on the JTAG chain, its position in the chain is defined by this option.
Devices Before	Specify the number of JTAG data bits before the device in the JTAG chain.
Devices After	Specify the number of JTAG data bits after the device in the JTAG chain.
Instruction bits Before	Specify the number of JTAG instruction register bits before the device in the JTAG chain.
Instruction bits After	Specify the number of JTAG instruction register bits after the device in the JTAG chain.

Download control

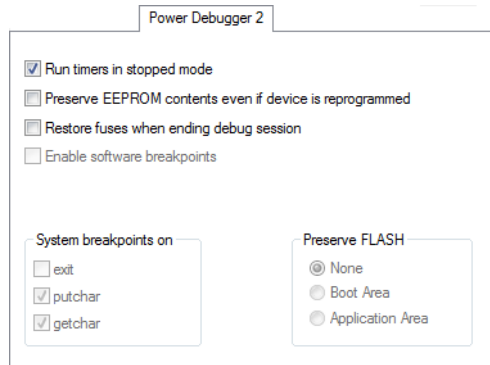
Controls the download.

Suppress download	<p>Disables the downloading of code, while preserving the present content of the flash memory. This command is useful if you want to debug an application that already resides in target memory. Note that the implicit RESET performed by C-SPY at startup is not disabled.</p> <p>If this option is combined with the Verify all option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.</p>
--------------------------	--

Allow download to RAM	Downloads constant data into RAM. By default, the option is deselected and an error message is displayed if you try to download constant data to RAM.
Target consistency check	<p>Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Choose between:</p> <p>None, target consistency check is not performed.</p> <p>Verify Boundaries, verifies the boundaries of each downloaded module. This is a fast and simple, but not complete, check of the memory.</p> <p>Verify All, checks every byte after loading. This is a slow, but complete, check of the memory.</p>

Power Debugger 2

The **Power Debugger 2** options control the C-SPY driver for Power Debugger.



Run timers in stopped mode

Runs the timers even if the program is stopped.

Preserve EEPROM contents even if device is reprogrammed

Erases flash memory before download. If this option is deselected, both the EEPROM and the flash memory will be erased when you reprogram the flash memory.

Restore fuses when ending debug session

Enables C-SPY to modify the OCD enable fuse and preserve the EEPROM fuse at startup. Note that each change of fuse switch will decrease the life span of the chip.

Enable software breakpoints

Enables the use of software breakpoints. The number of software breakpoints is unlimited. For more information about breakpoints, see *Breakpoints*, page 105.

System breakpoints on

Disables the use of system breakpoints in the CLIB runtime environment. If you do not use the C-SPY **Terminal I/O** window or if you do not need a breakpoint on the `exit` label, you can save hardware breakpoints by not reserving system breakpoints.

Select or deselect the options **exit**, **putchar**, and **getchar**, respectively.

In the DLIB runtime environment, C-SPY will always set a system breakpoint on the `__DebugBreak` label. You cannot disable this behavior.

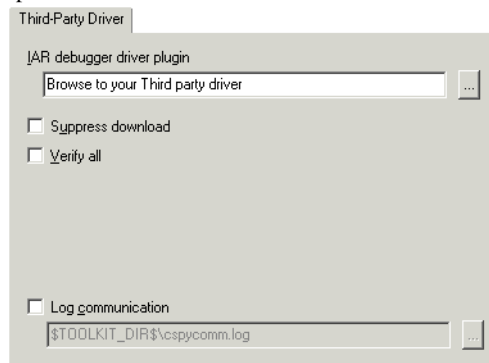
For more information, see *Breakpoint consumers*, page 110.

Preserve FLASH

Selects which part of the flash memory, if any, that you want to preserve during download. Choose between **None**, **Boot Area**, or **Application Area**.

Third-Party Driver options

The **Third-Party Driver** options are used for loading any driver plugin provided by a third-party vendor. These drivers must be compatible with the C-SPY debugger driver specification.



IAR debugger driver plugin

Specify the file path to the third-party driver plugin DLL file. A browse button is available for your convenience.

Suppress download

Disables the downloading of code, while preserving the present content of the flash. This command is useful if you need to exit C-SPY for a while and then continue the debug session without downloading code. The implicit RESET performed by C-SPY at startup is not disabled though.

If this option is combined with **Verify all**, the debugger will read your application back from the flash memory and verify that it is identical with the application currently being debugged.

This option can be used if it is supported by the third-party driver.

Verify all

Verifies that the memory on the target system is writable and mapped in a consistent way. A warning message will appear if there are any problems during download. Every byte is checked after it is loaded. This is a slow but complete check of the memory. This option can be used if is supported by the third-party driver.

Log communication

Logs the communication between C-SPY and the target system to a file. To interpret the result, detailed knowledge of the interface is required. This option can be used if is supported by the third-party driver.

Additional information on C-SPY drivers

This chapter describes the additional menus and features provided by the C-SPY® drivers. You will also find some useful hints about resolving problems.

Reference information on C-SPY driver menus

Reference information about:

- *C-SPY driver*, page 367
- *Simulator menu*, page 368
- *JTAGICE mkII menu*, page 369
- *Dragon menu*, page 369
- *Atmel-ICE menu*, page 370
- *JTAGICE3 menu*, page 371
- *AVR ONE! menu*, page 372
- *Power Debugger menu*, page 373

C-SPY driver

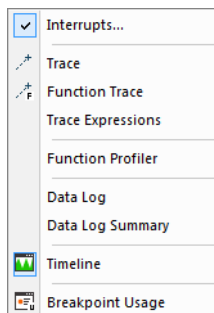
Before you start the C-SPY debugger, you must first specify a C-SPY driver in the **Options** dialog box, using the option **Debugger>Setup>Driver**.

When you start a debug session, a menu specific to that C-SPY driver will appear on the menu bar, with commands specific to the driver.

When we in this guide write “choose *C-SPY driver*>” followed by a menu command, *C-SPY driver* refers to the menu. If the feature is supported by the driver, the command will be on the menu.

Simulator menu

When you use the simulator driver, the **Simulator** menu is added to the menu bar.



Menu commands

These commands are available on the menu:

Interrupts

Displays a dialog box where you can manage interrupts, see *Interrupts dialog box*, page 248.



Trace

Opens a window which displays the collected trace data, see *Trace window*, page 164.



Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 167.

Trace Expressions

Opens a window where you can specify specific variables and expressions for which you want to collect trace data, see *Trace Expressions window*, page 170.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 202.

Data Log

Opens a window which logs accesses to up to four different memory locations or areas, see *Data Log window*, page 188.

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 191.



Timeline

Opens a window which gives a graphical view of various kinds of information on a timeline, see *The application timeline*, page 175.

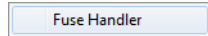


Breakpoint Usage

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 119.

JTAGICE mkII menu

When you are using the C-SPY JTAGICE mkII driver, the **JTAGICE mkII** menu is added to the menu bar. Before the debugger is started, the menu looks like this:

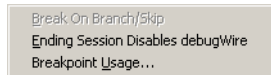


This command is available on the menu:

Fuse Handler

Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 375.

When the debugger is running, the menu looks like this:



These commands are available on the menu:

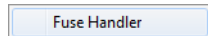
Break On Branch/Skip Stops execution just after each branch instruction.

Ending Session Disables debugWire Disables the use of debugWIRE before ending the debug session.

Breakpoint Usage Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 119.

Dragon menu

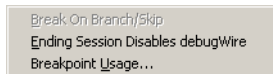
When you are using the C-SPY Dragon driver, the **Dragon** menu is added to the menu bar. Before the debugger is started, the menu looks like this:



This command is available on the menu:

Fuse Handler Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 375.

When the debugger is running, the menu looks like this:



These commands are available on the menu:

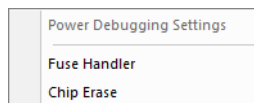
Break On Branch/Skip Stops execution just after each branch instruction.

Ending Session Disables debugWire Disables the use of debugWIRE before ending the debug session.

Breakpoint Usage Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 119.

Atmel-ICE menu

When you are using the C-SPY Atmel-ICE driver, the **Atmel-ICE** menu is added to the menu bar. Before the debugger is started, the menu looks like this:



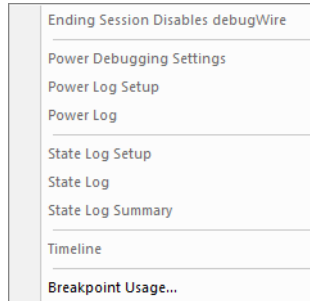
These commands are available on the menu:

Power Debugging Settings This command is not applicable to Atmel-ICE.

Fuse Handler Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 377.

Chip Erase Performs a chip erase, that is, erases flash memory, EEPROM, and lock bits. For more information, see the documentation for the target you are using.

When the debugger is running, the menu looks like this:

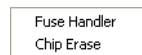


This command is available on the menu:

Ending Session Disables debugWire	Disables the use of debugWire before ending the debug session.
Power Debugging Settings	This command is not applicable to Atmel-ICE.
Power Log Setup	This command is not applicable to Atmel-ICE.
Power Log	This command is not applicable to Atmel-ICE.
State Log Setup	This command is not applicable to Atmel-ICE.
State Log	This command is not applicable to Atmel-ICE.
State Log Summary	This command is not applicable to Atmel-ICE.
Timeline	This command is not applicable to Atmel-ICE.
Breakpoint Usage	Opens a window which lists all active breakpoints, see <i>Breakpoint Usage window</i> , page 119.

JTAGICE3 menu

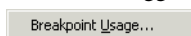
When you are using the C-SPY JTAGICE3 driver, the **JTAGICE3** menu is added to the menu bar. Before the debugger is started, the menu looks like this:



These commands are available on the menu:

- | | |
|---------------------|---|
| Fuse Handler | Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see <i>Fuse Handler dialog box</i> , page 377. |
| Chip Erase | Performs a chip erase, that is, erases flash memory, EEPROM, and lock bits. For more information, see the documentation for the target you are using. |

When the debugger is running, the menu looks like this:

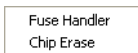


This command is available on the menu:

- | | |
|-------------------------|---|
| Breakpoint Usage | Opens a window which lists all active breakpoints, see <i>Breakpoint Usage window</i> , page 119. |
|-------------------------|---|

AVR ONE! menu

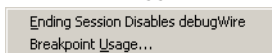
When you are using the C-SPY AVR ONE! driver, the **AVR ONE!** menu is added to the menu bar. Before the debugger is started, the menu looks like this:



These commands are available on the menu:

- | | |
|---------------------|---|
| Fuse Handler | Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see <i>Fuse Handler dialog box</i> , page 377. |
| Chip Erase | Performs a chip erase, that is, erases flash memory, EEPROM, and lock bits. For more information, see the documentation for the target you are using. |

When the debugger is running, the menu looks like this:



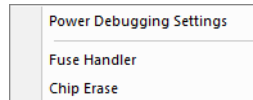
These commands are available on the menu:

- | | |
|--|--|
| Ending Session Disables debugWire | Disables the use of debugWIRE before ending the debug session. |
|--|--|

Breakpoint Usage Opens a window which lists all active breakpoints, see *Breakpoint Usage window*, page 119.

Power Debugger menu

When you are using the C-SPY Power Debugger driver, the Power Debugger menu is added to the menu bar. Before the debugger is started, the menu looks like this:



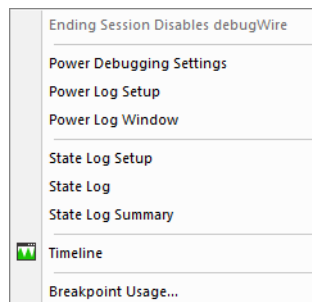
These commands are available on the menu:

Power Debugging Settings Displays a dialog box where you can select which GPIO input pins to monitor, see *Power Debugging Settings*, page 222.

Fuse Handler Displays a dialog box, which provides programming possibilities of the device-specific on-chip fuses, see *Fuse Handler dialog box*, page 377.

Chip Erase Performs a chip erase, that is, erases flash memory, EEPROM, and lock bits. For more information, see the documentation for the target you are using.

When the debugger is running, the menu looks like this:



These commands are available on the menu:

Ending Session Disables debugWire Disables the use of debugWire before ending the debug session.

Power Debugging Settings Displays a dialog box; see *Power Debugging Settings*, page 222.

Power Log Setup	Opens a window; see <i>Power Log Setup window</i> , page 220.
Power Log	Opens a window; see <i>Power Log window</i> , page 226.
State Log Setup	Opens a window; see <i>State Log Setup window</i> , page 229.
State Log	Opens a window; see <i>State Log window</i> , page 231.
State Log Summary	Opens a window; see <i>State Log Summary window</i> , page 233.
Timeline	Opens the Timeline window; see <i>Reference information on application timeline</i> , page 180.
Breakpoint Usage	Opens a window which lists all active breakpoints, see <i>Breakpoint Usage window</i> , page 119.

Reference information on the C-SPY hardware debugger drivers

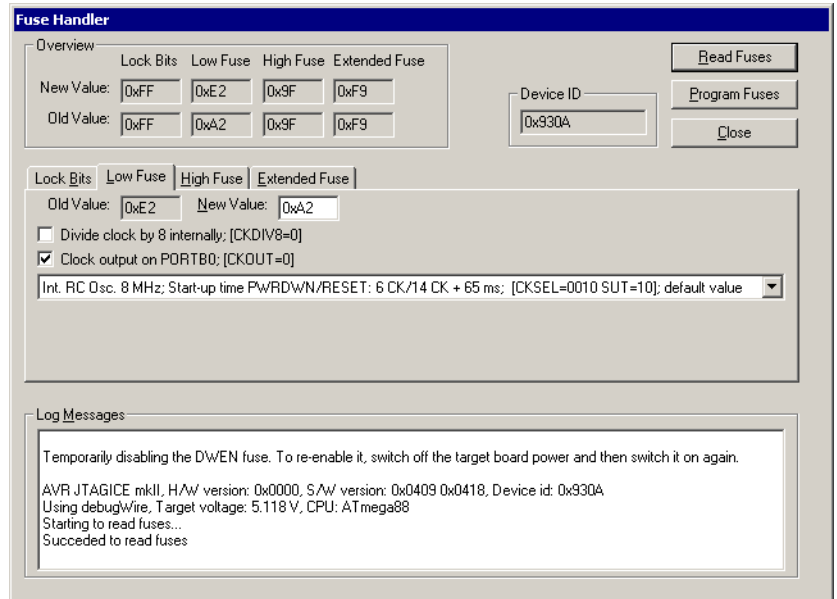
This section gives additional reference information on the C-SPY hardware debugger drivers, reference information not provided elsewhere in this documentation.

Reference information about:

- *Fuse Handler dialog box*, page 375 (JTAGICE mkII and Dragon)
- *Fuse Handler dialog box*, page 377 (Atmel-ICE, Power Debugger, AVR ONE!, and JTAGICE3)

Fuse Handler dialog box

The **Fuse Handler** dialog box is available from the **JTAGICE mkII** menu or the **Dragon** menu, respectively.



Note: To use the fuse handler, the JTAG Enable fuse must be enabled on one of the pages. If a debugWIRE interface is used, it will be temporarily disabled while using the fuse handler. Before you start debugging and after programming the fuses, you must enable the interface again. The JTAGICE mkII/Dragon debugger driver will guide you through this.

The fuse handler provides programming possibilities of the device-specific on-chip fuses and lock bits via JTAGICE mkII/Dragon.

Because different devices have different features, the available options and possible settings depend on which device is selected. However, the available options are divided into a maximum of four groups: **Lock Bits**, **Low Fuse**, **High Fuse**, and **Extended Fuse**. For detailed information about the fuse settings, see the device-specific documentation provided by Atmel® Corporation.

When you open the **Fuse Handler** dialog box, the device-specific fuses are read and the dialog box will reflect the fuse values.

Requirements

One of these alternatives:

- The C-SPY JTAGICE mkII driver
- The C-SPY Dragon driver.

Overview

Displays an overview of the fuse settings for each fuse group.

New Value Displays the value of the fuses reflecting the user-defined settings. In other words, the value of the fuses after they have been programmed.

Old Value Displays the last read value of the fuses.

Lock Bits, Low Fuse, High Fuse, Extended Fuse pages

Contain the available options for each group of fuses. The options and possible settings depend on the selected device.

Old Value Displays the last read value of the on-chip fuses on the device.

New Value Displays the value of the fuses reflecting the user-defined settings. This text field is editable.

To specify the fuse settings, you can either use the **New Value** text field or use the options.

Selecting an option means that this fuse should be enabled/programmed, which means that the on-chip fuse is set to zero.

Device ID

Displays the device ID that has been read from the device.

Read Fuses

Reads the on-chip fuses from the device and the **Before** text fields will be updated accordingly.

Program Fuses

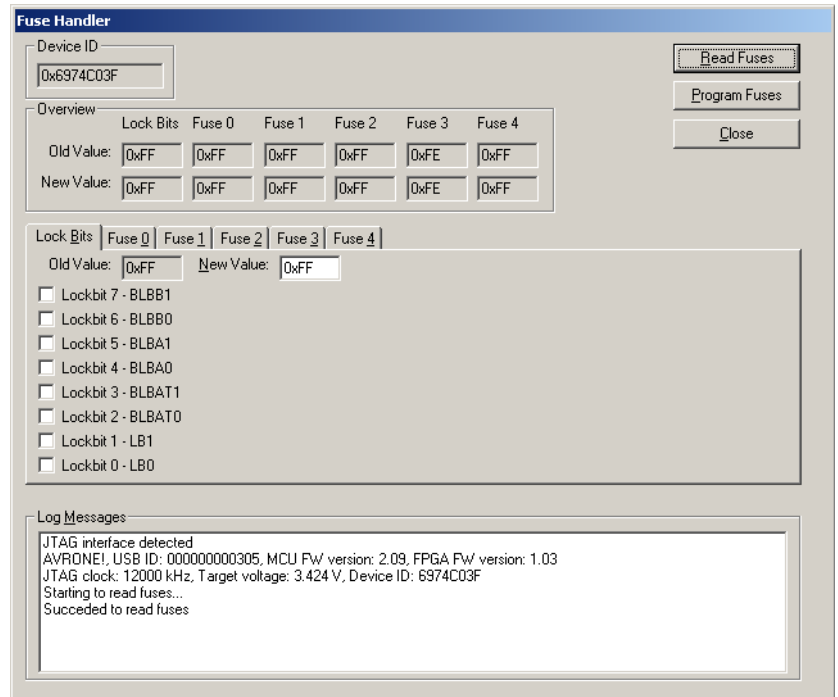
Programs the new fuse values—displayed in the **After** text box—to the on-chip fuses.

Log Messages

Displays the device information, and status and diagnostic messages for all read/program operations.

Fuse Handler dialog box

The **Fuse Handler** dialog box is available from the **Atmel-ICE** menu, the **Power Debugger** menu, the **AVR ONE!** menu, or the **JTAGICE3** menu, respectively.



Note: To use the fuse handler, the JTAG Enable fuse must be enabled on one of the tabs or you can use PDI, see *Atmel-ICE 1*, page 342, *Power Debugger 1*, page 362, *AVR ONE! 1*, page 348, or *JTAGICE3 1*, page 351, specifically the information about the **Debug Port** option. The debugger driver will guide you through this.

The fuse handler provides programming possibilities of the device-specific on-chip fuses and lock bits via the debug probe.

Because different devices have different features, the available options and possible settings depend on which device is selected. However, the available options are divided into a maximum of six groups: **Lock Bits**, **Fuse 0**, **Fuse 1**, **Fuse 2**, **Fuse 3**, and **Fuse 4**.

For detailed information about the fuse settings, see the device-specific documentation provided by Atmel® Corporation.

When you open the **Fuse Handler** dialog box, the device-specific fuses are read and the dialog box will reflect the fuse values.

Requirements

One of these alternatives:

- The C-SPY Atmel-ICE driver
- The C-SPY Power Debugger driver
- The C-SPY AVR ONE! driver
- The C-SPY JTAGICE3 driver.

Device ID

Displays the device ID that has been read from the device.

Overview

Displays an overview of the fuse settings for each fuse group.

New Value	Displays the value of the fuses reflecting the user-defined settings. In other words, the value of the fuses after they have been programmed.
Old Value	Displays the last read value of the fuses.

Lock Bits, Fuse 0, Fuse 1, Fuse 2, Fuse 3, and Fuse 4 pages

Contain the available options for each group of fuses. The options and possible settings depend on the selected device.

Old Value	Displays the last read value of the on-chip fuses on the device.
New Value	Displays the value of the fuses reflecting the user-defined settings. This text field is editable.

To specify the fuse settings, you can either use the **New Value** text field or use the options.

Selecting an option means that this fuse should be enabled/programmed, which means that the on-chip fuse is set to zero.

Read Fuses

Reads the on-chip fuses from the device and the **Before** text fields will be updated accordingly.

Program Fuses

Programs the new fuse values—displayed in the **After** text box—to the on-chip fuses.

Log Messages

Displays the device information, and status and diagnostic messages for all read/program operations.

Resolving problems

These topics are covered:

- No contact with the target hardware

Debugging using the C-SPY hardware debugger systems requires interaction between many systems, independent from each other. For this reason, setting up this debug system can be a complex task. If something goes wrong, it might be difficult to locate the cause of the problem.

This section includes suggestions for resolving the most common problems that can occur when debugging with the C-SPY hardware debugger systems.

For problems concerning the operation of the evaluation board, refer to the documentation supplied with it, or contact your hardware distributor.

NO CONTACT WITH THE TARGET HARDWARE

There are several possible reasons for C-SPY to fail to establish contact with the target hardware. Do this:

- Check the communication devices on your host computer
- Verify that the cable is properly plugged in and not damaged or of the wrong type
- Make sure that the evaluation board is supplied with sufficient power
- Check that the correct options for communication have been specified in the IAR Embedded Workbench IDE.

Examine the linker configuration file to make sure that the application has not been linked to the wrong address.

A

A access (Complex breakpoints option) 128
 Abort (Report Assert option) 81
 __abortLaunch (C-SPY system macro). 267
 absolute location, specifying for a breakpoint. 130
 Action (Data breakpoints option) 129
 Add to Watch Window (Symbolic Memory window context menu) 148
 Address Range (Find in Trace option) 173
 Allow download to RAM (Atmel-ICE option) 344, 364
 Allow download to RAM (AVR ONE! option) 349
 Allow download to RAM (Dragon option) 361
 Allow download to RAM (JTAGICE mkII option). 356
 Allow download to RAM (JTAGICE3 option). 353
 Ambiguous symbol (Resolve Symbol Ambiguity option). 104
 application, built outside the IDE 52
 assembler labels, viewing 89
 assembler source code, fine-tuning 197
 assembler symbols, using in C-SPY expressions 87
 assembler variables, viewing 89
 assumptions, programming experience. 21
 Atmel Power Debugger. 212
 Atmel-ICE options 342, 345
 Atmel-ICE (C-SPY driver). 37
 menu 370
 Auto Scroll (Timeline window context menu) 183, 186, 224, 237
 Auto window 91
 Autostep settings dialog box. 82
 Autostep (Debug menu) 58
 AVR ONE! options. 348, 350
 AVR ONE! (C-SPY driver) 45
 hardware installation 46
 menu 372
 --avrone_jtag_clock (C-SPY command line option) 316

B

B access (Complex breakpoints option) 128
 --attach_to_running_target (C-SPY command line option) 315
 --backend (C-SPY command line option). 316
 backtrace information
 viewing in Call Stack window 75
 batch mode, using C-SPY in 309
 Baud (debugger option) 358
 Big Endian (Memory window context menu) 142
 blocks, in C-SPY macros 261
 bold style, in this guide 25
 Break On Branch/Skip (Dragon menu) 370
 Break On Branch/Skip (JTAGICE mkII menu) 369
 Break on Throw (Debug menu) 59
 Break on Uncaught Exception (Debug menu). 59
 Break (Debug menu). 58
 breakpoint condition, example 115–116
 Breakpoint control (Complex breakpoints option) 128
 Breakpoint Usage window 119
 Breakpoint Usage (Atmel-ICE menu). 371
 Breakpoint Usage (AVR ONE! menu) 373
 Breakpoint Usage (Dragon menu). 370
 Breakpoint Usage (JTAGICE mkII menu) 369
 Breakpoint Usage (JTAGICE3 menu) 372
 Breakpoint Usage (Power Debugger menu) 374
 breakpoints
 code, example 286
 complex 127
 example. 289
 connecting a C-SPY macro 256
 consumers of 110
 data 123
 data log 125
 description of. 105
 disabling used by Stack window 111
 icons for in the IDE 108
 in Memory window 113
 listing all 119

reasons for using	105
setting	
in memory window	113
using system macros	114
using the dialog box	112
single-stepping if not available	50
toggleing	112
types of	106
useful tips	115
Breakpoints dialog box	
Code	120
Complex	127
Data	123
Data Log	125
Immediate	126
Log	121
Trace Start	168
Trace Stop	169
Breakpoints window	117
Browse (Trace toolbar)	164
byte order, setting in Memory window	142

C

C function information, in C-SPY	68
C symbols, using in C-SPY expressions	86
C variables, using in C-SPY expressions	86
Calibrate (Power Debugging Settings option)	222
call chain, displaying in C-SPY	68
Call stack information	68
Call Stack window	75
for backtrace information	68
Call Stack (Timeline window context menu)	183
__cancelAllInterrupts (C-SPY system macro)	268
__cancelInterrupt (C-SPY system macro)	268
Chip Erase (Atmel-ICE menu)	370
Chip Erase (JTAGICE3 menu)	372
Chip Erase (Power Debugger menu)	373
Clear All (Debug Log window context menu)	80

Clear Group (Registers User Groups Setup window context menu)	157
Clear Group (Registers User Groups Setup window context menu)	157
Clear trace data (Trace toolbar)	164
Clear (Power Log window context menu)	228
Clear (State Log Summary window context menu)	235
Clear (State Log window context menu)	232
__clearBreak (C-SPY system macro)	269
CLIB	
consuming breakpoints	110
library reference information for	24
naming convention	26
__closeFile (C-SPY system macro)	269
code breakpoints	
overview	106
toggleing	112
Code Coverage window	208
Code Coverage (Disassembly window context menu)	73
--code_coverage_file (C-SPY command line option)	317
code, covering execution of	208
command line options	315
typographic convention	25
command prompt icon, in this guide	25
Communication (JTAGICE mkII option)	355
complex breakpoints, overview	107
Complex data (Complex breakpoints option)	128
computer style, typographic convention	25
conditional statements, in C-SPY macros	260
context menu, in windows	89
conventions, used in this guide	24
Copy Window Contents (Disassembly window context menu)	75
Copy (Debug Log window context menu)	80
copyright notice	2
Core (Cores window)	83
cores	
inspecting state of	82
Cores window	82
--cpu (C-SPY command line option)	317

- cspybat 309
 - reading options from file (-f) 326
 - current position, in C-SPY Disassembly window 72
 - cursor, in C-SPY Disassembly window 72
 - cycles (C-SPY command line option) 318
 - Cycles (Cores window) 83
 - C-SPY
 - batch mode, using in 309
 - debugger systems, overview of 33
 - differences between drivers 35
 - environment overview 29
 - plugin modules, loading 51
 - setting up 49–50
 - starting the debugger 51
 - C-SPY drivers
 - Atmel-ICE 37
 - AVR ONE! 45
 - differences between 35
 - Dragon 43
 - JTAGICE mkII 43
 - JTAGICE3 41
 - overview 35
 - Power Debugger 39
 - specifying 339
 - types of 34
 - C-SPY expressions 86
 - evaluating, using Macro Quicklaunch window 306
 - evaluating, using Quick Watch window 100
 - in C-SPY macros 260
 - Tooltip watch, using 85
 - Watch window, using 85
 - C-SPY macros
 - blocks 261
 - conditional statements 260
 - C-SPY expressions 260
 - examples 253
 - checking status of register 255
 - creating a log macro 256
 - executing 253
 - connecting to a breakpoint 256
 - using Quick Watch 255
 - using setup macro and setup file 255
 - functions 87, 258
 - keywords 258–259, 261
 - loop statements 261
 - macro statements 260
 - parameters 259
 - setup macro file 252
 - executing 255
 - setup macro functions 252
 - summary 263
 - system macros, summary of 265
 - using 251
 - variables 88, 258
 - C-SPY options
 - Extra Options 347
 - Images 340
 - Plugins 341
 - Setup 339
 - C-SPYLink 34
 - C-STAT for static analysis, documentation for 23
 - C++ terminology 24
- ## D
- Data bits (debugger option) 358
 - data breakpoints, overview 106
 - Data Coverage (Memory window context menu) 142
 - data coverage, in Memory window 141
 - data log breakpoints, overview 107
 - Data Log Summary window 191
 - Data Log Summary (Simulator menu) 368
 - Data Log window 188
 - Data Log (Simulator menu) 368
 - ddf (filename extension), selecting a file 51
 - Debug Log window 79
 - Debug menu (C-SPY main window) 57

Debug Port (Atmel-ICE option)	343	__disableInterrupts (C-SPY system macro)	270
Debug Port (AVR ONE! option).	348	--disable_internal_eeprom (C-SPY command line option)	319
Debug Port (JTAGICE3 option)	351	--disable_interrupts (C-SPY command line option)	319
Debug Port (Power Debugger option).	362	Disassembly window	71
Debug (Report Assert option)	81	context menu	73
--debugfile (cspybat option)	318	disclaimer	2
debugger concepts, definitions of	32	DLIB	
debugger drivers		consuming breakpoints	110
simulator	36	naming convention.	26
Debugger Macros window	304	do (macro statement)	261
debugger system overview	33	document conventions	24
debugging projects		documentation	
externally built applications	52	overview of guides.	23
loading multiple images	53	overview of this guide	21
debugging, RTOS awareness	31	Download control (Atmel-ICE option)	344
debugWIRE	369–370	Download control (AVR ONE! option)	349
Default communication (Serial Port option)	358	Download control (Dragon option).	360
__delay (C-SPY system macro)	269	Download control (JTAGICE mkII option)	356
Delay (Autostep Settings option)	82	Download control (JTAGICE3 option).	352
Delete (Breakpoints window context menu)	118	Download control (Power Debugger option)	363
Device description file (debugger option)	340	--download_only (C-SPY command line option)	319
device description files	51	Dragon options	359, 361
definition of	55	Dragon (C-SPY driver).	43
memory zones	135	menu	369
modifying	55	Driver (debugger option)	339
register zone.	135	__driverType (C-SPY system macro)	270
Device ID (Fuse Handler option)	376, 378	--drv_communication (C-SPY command line option).	319
Devices After (Atmel-ICE option)	343, 363	--drv_communication_log (C-SPY command line option)	320
Devices After (AVR ONE! option).	349	--drv_debug_port (C-SPY command line option)	320
Devices After (Dragon option)	360	--drv_download_data (C-SPY command line option).	321
Devices After (JTAGICE mkII option).	355	--drv_dragon (C-SPY command line option)	321
Devices After (JTAGICE3 option)	352	--drv_power_debugger (C-SPY command line option).	322
Devices Before (Atmel-ICE option)	343, 363	--drv_preserve_app_section (C-SPY command line option)	322
Devices Before (AVR ONE! option)	349	--drv_preserve_boot_section (C-SPY command line option)	322
Devices Before (Dragon option)	360	--drv_set_exit_breakpoint (C-SPY command line option)	323
Devices Before (JTAGICE mkII option)	355	--drv_set_getchar_breakpoint (C-SPY command line option)	323
Devices Before (JTAGICE3 option)	352		
Disable All (Breakpoints window context menu)	118		
Disable (Breakpoints window context menu)	118		

--drv_set_putchar_breakpoint
(C-SPY command line option) 324
 --drv_suppress_download (C-SPY command line option) 324
 --drv_use_PDI (C-SPY command line option) 325
 --drv_verify_download (C-SPY command line option) . . 325

E

Edit Expressions (Trace toolbar) 165
 Edit Settings (Trace toolbar) 165
 Edit (Breakpoints window context menu) 118
 edition, of this guide 2
 EEPROM
 contents, preserving in Atmel-ICE 345
 contents, preserving in AVR ONE! 350
 contents, preserving in Dragon 361
 contents, preserving in JTAGICE mkII 357
 contents, preserving in JTAGICE3 353
 contents, preserving in Power Debugger 364
 --eeprom_size (C-SPY command line option) 325
 Enable All (Breakpoints window context menu) 118
 Enable Log File (Log File option) 80
 Enable simulation (Interrupts option) 248
 Enable software breakpoints (Atmel-ICE option) 345
 Enable software breakpoints (AVR ONE! option) 350
 Enable software breakpoints (Dragon option) 362
 Enable software breakpoints (JTAGICE mkII option) . . . 357
 Enable software breakpoints (JTAGICE3 option) 354
 Enable software breakpoints (Power Debugger option) . . 365
 Enable (Breakpoints window context menu) 118
 Enable (Power Log window context menu) 228
 Enable (State Log Summary window context menu) . . . 235
 Enable (State Log window context menu) 232
 Enable (Timeline window context menu) 183
 Enabled GPIO inputs (Power Debugging Settings option) 222
 __enableInterrupts (C-SPY system macro) 271
 Enable/Disable Breakpoint (Call
 Stack window context menu) 77
 Enable/Disable Breakpoint (Disassembly window context
 menu) 74

Enable/Disable (Trace toolbar) 164
 End address (Memory Save option) 144
 endianness. *See* byte order
 Ending Session Disables debugWire (AVR ONE! menu) . 372
 Ending Session Disables debugWIRE (Dragon menu) . . . 370
 Ending Session Disables debugWire
 (Power Debugger menu) 373
 Ending Session Disables
 debugWIRE (JTAGICE mkII menu) 369
 --enhanced_core (C-SPY command line option) 326
 Enter Location dialog box 130
 __evaluate (C-SPY system macro) 271
 Evaluate Now (Macro Quicklaunch
 window context menu) 307
 examples
 C-SPY macros 253
 interrupts
 timer 246
 macros
 checking status of register 255
 creating a log macro 256
 using Quick Watch 255
 performing tasks and continue execution 116
 tracing incorrect function arguments 115
 execUserExecutionStarted (C-SPY setup macro) 264
 execUserExecutionStopped (C-SPY setup macro) 264
 execUserExit (C-SPY setup macro) 265
 execUserPreload (C-SPY setup macro) 263
 execUserPreReset (C-SPY setup macro) 265
 execUserReset (C-SPY setup macro) 265
 execUserSetup (C-SPY setup macro) 264
 executed code, covering 208
 execution history, tracing 163
 Execution state (Cores window) 82
 expressions. *See* C-SPY expressions
 extended command line file, for cspybat 326
 Extra Options, for C-SPY 347

F	
-f (cspybat option)	326
File format (Memory Save option)	144
file types	
device description, specifying in IDE	51
macro	50, 339
filename extensions	
ddf, selecting device description file	51
mac, using macro file	50
Filename (Memory Restore option)	145
Filename (Memory Save option)	144
Fill dialog box	145
__writeMemory8 (C-SPY system macro)	272
__writeMemory16 (C-SPY system macro)	272
__writeMemory32 (C-SPY system macro)	273
Find in Trace dialog box	172
Find in Trace window	173
Find in Trace (Disassembly window context menu)	75
Find (Memory window context menu)	142
Find (Trace toolbar)	165
first activation time (interrupt property)	
definition of	244
flash memory, load library module to	276
__fmessage (C-SPY macro keyword)	261
for (macro statement)	261
Format (Registers User Groups	
Setup window context menu)	157
Format (Registers window context menu)	155
formats, C-SPY input	31
Frequency in Hz (Atmel-ICE option)	343, 363
Frequency in Hz (AVR ONE! option)	348
Frequency in Hz (Dragon option)	360
Frequency in Hz (JTAGICE mkII option)	355
Frequency in Hz (JTAGICE3 option)	352
Function Profiler window	202
Function Profiler (Simulator menu)	368
Function Trace window	167
functions	
C-SPY running to when starting	50, 339
most time spent in, locating	197
--function_profiling (cspybat option)	326
Fuse Handler dialog box	375, 377
Fuse Handler (Atmel-ICE menu)	370
Fuse Handler (AVR ONE! menu)	372
Fuse Handler (Dragon menu)	370
Fuse Handler (JTAGICE mkII menu)	369
Fuse Handler (JTAGICE3 menu)	372
Fuse Handler (Power Debugger menu)	373
G	
__getCycleCounter (C-SPY system macro)	274
Go to Source (Breakpoints window context menu)	118
Go to Source (Call Stack window context menu)	76
Go To Source (Timeline window	
context menu)	183, 187, 225, 238
Go (Debug menu)	57, 67
H	
Handshaking (debugger option)	359
Hardware reset on C-SPY reset (AVR ONE! option)	350
Hardware reset on C-SPY reset (Dragon option)	362
Hardware reset on C-SPY reset (JTAGICE mkII	
option)	357
hardware setup, power consumption because of	216
highlighting, in C-SPY	68
I	
IAR debugger driver plugin (debugger option)	366
icons, in this guide	25
if else (macro statement)	260
if (macro statement)	260
Ignore (Report Assert option)	81
Images window	60

Images, loading multiple 340
 immediate breakpoints, overview 107
 Include (Log File option) 80
 input formats, C-SPY 31
 Input Mode dialog box 78
 input, special characters in Terminal I/O window 78
 installation directory 24
 Instruction bits After (Atmel-ICE option) 343, 363
 Instruction bits After (AVR ONE! option) 349
 Instruction bits After (Dragon option) 360
 Instruction bits After (JTAGICE mkII option) 356
 Instruction bits After (JTAGICE3 option) 352
 Instruction bits Before (Atmel-ICE option) 343, 363
 Instruction bits Before (AVR ONE! option) 349
 Instruction bits Before (Dragon option) 360
 Instruction bits Before (JTAGICE mkII option) 356
 Instruction bits Before (JTAGICE3 option) 352
 Instruction Profiling (Disassembly window context menu) 74
 Intel-extended, C-SPY input format 31
 Intel-extended, C-SPY output format 34
 interference, power consumption because of 217
 interrupt handling, power consumption during 215
 interrupt system, using device description file 245
 interrupts
 adapting C-SPY system for target hardware 245
 simulated, introduction to 243
 timer, example 246
 using system macros 245
 Interrupts dialog box 248
 __isBatchMode (C-SPY system macro) 275
 italic style, in this guide 25
 I/O register. *See* SFR

J

JTAG daisy chain (AVR ONE! option) 348
 JTAG daisy chain (Dragon option) 360
 JTAG daisy chain (JTAGICE mkII option) 355
 JTAG daisy chain (JTAGICE3 option) 352

JTAG Port (Atmel-ICE option) 343
 JTAG Port (AVR ONE! option) 348
 JTAG Port (Dragon option) 360
 JTAG Port (JTAGICE mkII option) 355
 JTAG Port (JTAGICE3 option) 352
 JTAG Port (Power Debugger option) 363
 JTAGICE mkII options 354, 357
 JTAGICE mkII (C-SPY driver) 43
 hardware installation 44
 menu 369
 --jtagicemkII_use_software_breakpoints
 (C-SPY command line option) 329
 --jtagice_clock (C-SPY command line option) 327
 --jtagice_do_hardware_reset
 (C-SPY command line option) 327
 --jtagice_leave_timers_running
 (C-SPY command line option) 328
 --jtagice_preserve_eeprom (C-SPY command line option) 328
 --jtagice_restore_fuse (C-SPY command line option) . . . 328
 JTAGICE3 options 351, 353
 JTAGICE3 (C-SPY driver) 41
 hardware installation 38, 40, 42
 menu 371

L

labels (assembler), viewing 89
 --leave_target_running (C-SPY command line option) . . . 329
 Length (Fill option) 145
 library functions
 C-SPY support for using, plugin module 331
 online help for 24
 lightbulb icon, in this guide 25
 linker options
 typographic convention 25
 consuming breakpoints 110
 Little Endian (Memory window context menu) 142
 __loadImage (C-SPY system macro) 275
 loading multiple debug files, list currently loaded 60
 loading multiple images 53

Locals window	93
log breakpoints, overview	106
Log communication (debugger option)	346, 359, 366
Log File dialog box	80
Log Messages (Fuse Handler option)	377, 379
Logging>Set Log file (Debug menu)	59
Logging>Set Terminal I/O Log file (Debug menu)	59
loop statements, in C-SPY macros	261
low-power mode, power consumption during	214

M

mac (filename extension), using a macro file	50
--macro (C-SPY command line option)	330
macro files, specifying	50, 339
Macro Quicklaunch window	306
Macro Registration window	302
macro statements	260
macros	
executing	253
using	251
--macro-param (C-SPY command line option)	330
main function, C-SPY running to when starting	50, 339
Memory Fill (Memory window context menu)	142
Memory Restore dialog box	144
Memory Restore (Memory window context menu)	143
Memory Save dialog box	143
Memory Save (Memory window context menu)	143
Memory window	140
memory zones	134
in device description file	135
__memoryRestore (C-SPY system macro)	276
__memoryRestoreFromFile (C-SPY system macro)	277
__memorySave (C-SPY system macro)	277
__memorySaveToFile (C-SPY system macro)	278
Memory>Restore (Debug menu)	59
Memory>Save (Debug menu)	59
menu bar, C-SPY-specific	57
__message (C-SPY macro keyword)	261

__messageBoxYesCancel (C-SPY system macro)	279
__messageBoxYesNo (C-SPY system macro)	279
Messages window, amount of output	79
migration, from earlier IAR compilers	23
MISRA C	
documentation	23
Mixed Mode (Disassembly window context menu)	75
Motorola, C-SPY input format	31
Motorola, C-SPY output format	34
Move to PC (Disassembly window context menu)	73

N

naming conventions	25
Navigate (Timeline window context menu)	182, 186, 224, 237
New Breakpoint (Breakpoints window context menu)	118
Next Statement (Debug menu)	58
Next Symbol (Symbolic Memory window context menu)	148

O

Open Setup Window (Power Log window context menu)	229
Open Setup Window (Timeline window context menu)	226, 238
Open User Groups Setup Window (Registers window context menu)	155
__openFile (C-SPY system macro)	280
Operation (Fill option)	145
operators, sizeof in C-SPY	88
optimizations, effects on variables	88
options	
in the IDE	337
on the command line	315, 347
Options (Stack window context menu)	152
__orderInterrupt (C-SPY system macro)	281
Originator (debugger option)	341

- P**
- p (C-SPY command line option) 331
 - __param (C-SPY macro keyword) 259
 - parameters
 - tracing incorrect values of 68
 - typographic convention 25
 - Parity (debugger option) 358
 - part number, of this guide 2
 - PC (Cores window) 83
 - peripheral units
 - debugging power consumption for 211
 - detecting mistakenly unattended 215
 - detecting unattended 215
 - device-specific 55
 - displayed in Registers window 134
 - in an event-driven system 215
 - in C-SPY expressions 87
 - initializing using setup macros 252
 - peripherals register. *See* SFR
 - Please select one symbol
 - (Resolve Symbol Ambiguity option) 104
 - plugin (C-SPY command line option) 331
 - plugin modules (C-SPY) 34
 - loading 51
 - Plugins (C-SPY options) 341
 - pop-up menu. *See* context menu
 - Port (Serial Port option) 358
 - power consumption, measuring 198, 211
 - Power Debugger options 362, 364
 - Power Debugger (C-SPY driver) 39
 - Power Debugging Settings dialog box 222
 - Power Debugging Settings (Power Debugger menu) 373
 - Power Log Setup window 220
 - Power Log Setup (Power Debugger menu) 374
 - Power Log window 226
 - Power Log Window (Power Debugger menu) 374
 - Power Log (Timeline window context menu) 225
 - power sampling 198
 - prerequisites, programming experience 21
 - Preserve EEPROM contents (Atmel-ICE option) 345
 - Preserve EEPROM contents (AVR ONE! option) 350
 - Preserve EEPROM contents (Dragon option) 361
 - Preserve EEPROM contents (JTAGICE mkII option) 357
 - Preserve EEPROM contents (JTAGICE3 option) 353
 - Preserve EEPROM contents (Power Debugger option) 364
 - Preserve FLASH (Atmel-ICE option) 346
 - Preserve FLASH (AVR ONE! option) 351
 - Preserve FLASH (JTAGICE mkII option) 358
 - Preserve FLASH (JTAGICE3 option) 354
 - Preserve FLASH (Power Debugger option) 365
 - Previous Symbol (Symbolic
 - Memory window context menu) 148
 - probability (interrupt property)
 - definition of 244
 - Profile Selection (Timeline window
 - context menu) 184, 226, 239
 - profiling
 - analyzing data 199
 - on function level 199
 - on instruction level 201
 - profiling information, on functions and instructions 197
 - profiling sources
 - trace (calls) 198, 203
 - trace (flat) 198, 204
 - program execution
 - breaking 106–107
 - in C-SPY 63
 - Program Fuses (Fuse Handler option) 376, 379
 - programming experience 21
 - program_fuses_after_download
 - (C-SPY command line option) 332
 - program. *See* application
 - projects, for debugging externally built applications 52
 - publication date, of this guide 2

Q

Quick Watch window	100
executing C-SPY macros	255

R

Range for (Viewing Range option)	194
Read Fuses (Fuse Handler option)	376, 379
__readFile (C-SPY system macro)	282
__readFileByte (C-SPY system macro)	283
__readMemoryByte (C-SPY system macro)	283
__readMemory8 (C-SPY system macro)	283
__readMemory16 (C-SPY system macro)	284
__readMemory32 (C-SPY system macro)	284
reference information, typographic convention.	25
Refresh (Debug menu)	59
register groups	134
predefined, enabling.	153
Register User Groups Setup window	156
registered trademarks	2
__registerMacroFile (C-SPY system macro)	285
Registers window	153
registers, displayed in Registers window	153
Remove All (Macro Quicklaunch window context menu)	307
Remove (Macro Quicklaunch window context menu)	307
Remove (Registers User Groups Setup window context menu)	157
repeat interval (interrupt property), definition of	244
Replace (Memory window context menu)	142
Report Assert dialog box	81
Reset (Debug menu)	58
__resetFile (C-SPY system macro)	285
Resolve Source Ambiguity dialog box	131
Restore fuses when ending debug session (Atmel-ICE option)	345
Restore fuses when ending debug session (AVR ONE! option)	350

Restore fuses when ending debug session (JTAGICE3 option)	354
Restore fuses when ending debug session (Power Debugger option)	365
Restore (Memory Restore option).	145
return (macro statement).	261
ROM-monitor, definition of	34
RTOS awareness debugging.	31
RTOS awareness (C-SPY plugin module)	31
Run timers in stopped mode (Atmel-ICE option)	345
Run timers in stopped mode (AVR ONE! option)	350
Run timers in stopped mode (Dragon option)	361
Run timers in stopped mode (JTAGICE mkII option).	357
Run timers in stopped mode (JTAGICE3 option)	353
Run timers in stopped mode (Power Debugger option).	364
Run to Cursor (Call Stack window context menu)	76
Run to Cursor (Debug menu)	58
Run to Cursor (Disassembly window context menu)	73
Run to Cursor, command for executing	68
Run to (C-SPY option)	50, 339

S

Save to File (Register User Groups Setup window context menu)	157
Save to File (Registers window context menu).	155
Save to File (Timeline window context menu)	183
Save (Memory Save option)	144
Save (Trace toolbar)	165
Scale (Viewing Range option)	194
Select All (Debug Log window context menu).	80
Select Graphs (Timeline window context menu)	183, 187, 226, 238
Select plugins to load (debugger option).	341
serial port setup, hardware drivers	358
Set Data Breakpoint (Memory window context menu).	143
Set Data Log Breakpoint (Memory window context menu)	143
Set Next Statement (Debug menu)	59
Set Next Statement (Disassembly window context menu)	74

- __setCodeBreak (C-SPY system macro) 286
- __setComplexBreak (C-SPY system macro) 287
- __setDataBreak (C-SPY system macro) 289
- __setLogBreak (C-SPY system macro) 291
- __setSimBreak (C-SPY system macro) 292
- __setTraceStartBreak (C-SPY system macro) 293
- __setTraceStopBreak (C-SPY system macro) 294
- setup macro file, registering 50
- setup macro functions 252
 - reserved names 263
- Setup macros (debugger option) 339
- Setup (C-SPY options) 339
- SFR
 - in Registers window 155
 - using as assembler symbols 87
- shortcut menu. *See* context menu
- Show all images (Images window context menu) 61
- Show Arguments (Call Stack window context menu) 76
- Show Cycles (State Log Summary window context menu) 235
- Show Cycles (State Log window context menu) 233
- Show Numerical Value (Timeline window context menu) 187, 225
- Show offsets (Stack window context menu) 151
- Show only (Image window context menu) 61
- Show Time (State Log Summary window context menu) 235
- Show Time (State Log window context menu) 233
- Show variables (Stack window context menu) 151
- silent (C-SPY command line option) 333
- simulating interrupts, enabling/disabling 248
- Simulator menu. 368
- simulator, introduction 36
- Size (Timeline window context menu) 187, 225
- sizeof 88
- __smessage (C-SPY macro keyword) 261
- software breakpoints
 - enabling for Atmel-ICE 345
 - enabling for AVR Dragon 329, 362
 - enabling for AVR ONE! 329, 350
 - enabling for JTAGICE mkII 329, 357
 - enabling for JTAGICE3 329, 354
 - enabling for Power Debugger 365
 - use of 110
 - software delay, power consumption during 213
- Solid Graph (Timeline window context menu) 187, 225
- Sort by (Timeline window context menu) 238
- source (Timeline window context menu) 238
- __sourcePosition (C-SPY system macro) 295
- special function registers (SFR)
 - in Registers window 155
 - using as assembler symbols 87
- Stack window 149
- stack.mac 251
- standard C, sizeof operator in C-SPY 88
- Start address (Fill option) 145
- Start address (Memory Save option) 144
- State Log Setup (Power Debugger menu) 374
- State Log Summary window 233
- State Log Summary (Power Debugger menu) 374
- State Log window 231
- State Log (Power Debugger menu) 374
- State Log (Timeline window context menu) 238
- static analysis
 - documentation for 23
- Statics window 97
- Status (Cores window) 83
- stdin and stdout, redirecting to C-SPY window 77
- Step Into (Debug menu) 58
- Step Into, description 65
- Step Out (Debug menu) 58
- Step Out, description 66
- Step Over (Debug menu) 58
- Step Over, description 65
- step points, definition of 64
- Stop bits (debugger option) 359
- Stop Debugging (Debug menu) 58
- __strFind (C-SPY system macro) 295
- __subString (C-SPY system macro) 296

Suppress download	
Atmel-ICE option	344, 363
AVR ONE! option	349
debugger option	366
Dragon option	360
JTAGICE mkII option	356
JTAGICE3 option	352
Symbolic Memory window	146
Symbols window	102
symbols, using in C-SPY expressions	86
System breakpoints on (Atmel-ICE option)	345
System breakpoints on (AVR ONE! option)	350
System breakpoints on (Dragon option)	362
System breakpoints on (JTAGICE mkII option)	357
System breakpoints on (JTAGICE3 option)	354
System breakpoints on (Power Debugger option)	365

T

Target Consistency Check (Atmel-ICE option)	344, 364
Target Consistency Check (AVR ONE! option)	349
Target Consistency Check (Dragon option)	361
Target Consistency Check (JTAGICE mkII option)	356
Target Consistency Check (JTAGICE3 option)	353
Target device is part of a JTAG daisy chain (Atmel-ICE option)	343, 363
Target device is part of a JTAG daisy chain (AVR ONE! option)	348
Target device is part of a JTAG daisy chain (Dragon option)	360
Target device is part of a JTAG daisy chain (JTAGICE mkII option)	355
Target device is part of a JTAG daisy chain (JTAGICE3 option)	352
target system, definition of	33
__targetDebuggerVersion (C-SPY system macro)	296
Terminal IO Log Files (Terminal IO Log Files option)	78
Terminal I/O Log Files dialog box	78
Terminal I/O window	69, 77

terminology	24
Text search (Find in Trace option)	172
Third-Party Driver (debugger options)	365
Time Axis Unit (Timeline window context menu)	184, 187, 226, 239
Timeline window	181, 184, 229, 235
Timeline (Power Debugger menu)	374
--timeout (C-SPY command line option)	333
timer interrupt, example	246
timers (Atmel-ICE), running in stopped mode	345
timers (AVR ONE!), running in stopped mode	350
timers (Dragon), running in stopped mode	361
timers (JTAGICE mkII), running in stopped mode	357
timers (JTAGICE3), running in stopped mode	353
timers (Power Debugger), running in stopped mode	364
Toggle Breakpoint (Code) (Call Stack window context menu)	76
Toggle Breakpoint (Code) (Disassembly window context menu)	74
Toggle Breakpoint (Log) (Call Stack window context menu)	76
Toggle Breakpoint (Log) (Disassembly window context menu)	74
Toggle Breakpoint (Trace Start) (Call Stack window context menu)	77
Toggle Breakpoint (Trace Start) (Disassembly window context menu)	74
Toggle Breakpoint (Trace Stop) (Call Stack window context menu)	77
Toggle Breakpoint (Trace Stop) (Disassembly window context menu)	74
Toggle source (Trace toolbar)	164
__toLower (C-SPY system macro)	297
tools icon, in this guide	25
__toString (C-SPY system macro)	297
__toUpper (C-SPY system macro)	298
trace	161, 175
Trace Expressions window	170
trace start and stop breakpoints, overview	106
Trace Start breakpoints dialog box	168

Trace Stop breakpoints dialog box 169
 Trace window 164
 trace (calls), profiling source 198, 203
 trace (flat), profiling source 198, 204
 trace, in Timeline window 181, 184, 229, 235
 trademarks 2
 typographic conventions 25

U

UBROF. 31
 Unavailable, C-SPY message 89
 Universal Binary Relocatable Object Format. *See* UBROF
 __unloadImage(C-SPY system macro). 298
 USB
 Atmel-IVE option 346
 AVR ONE! option 346
 Dragon option 346
 JTAGICE mkII option 355
 JTAGICE3 option 346
 Power Debugger option 346
 USB ID (Atmel_ICE option) 346
 USB ID (AVR ONE! option) 346
 USB ID (Dragon option). 346
 USB ID (JTAGICE mkII option) 355
 USB ID (JTAGICE3 option). 346
 USB ID (Power Debugger option) 346
 Use command line options (debugger option). 347
 Use Extra Images (debugger option). 340
 Use PDI (JTAGICE mkII option). 355, 359
 Use UBROF reset vector (debugger option). 339
 user application, definition of 33

V

-v (C-SPY command line option) 333
 Value (Fill option) 145
 __var (C-SPY macro keyword). 258

variables
 effects of optimizations 88
 information, limitation on 88
 using in C-SPY expressions. 86
 variance (interrupt property), definition of 244
 Verify all (debugger option) 366
 version
 of this guide 2
 View Group (Registers window context menu) 155
 View User Group (Registers window context menu) 155
 Viewing Range dialog box 194
 Viewing Range (Timeline window context menu) . . 187, 225
 visualSTATE, C-SPY plugin module for 34

W

waiting for device, power consumption during. 213
 warnings icon, in this guide 25
 Watch window 95
 using 85
 web sites, recommended. 24
 while (macro statement) 261
 windows, specific to C-SPY 59
 With I/O emulation modules (linker option), using. 77
 __writeFile (C-SPY system macro) 299
 __writeFileByte (C-SPY system macro). 299
 __writeMemoryByte (C-SPY system macro) 300
 __writeMemory8 (C-SPY system macro). 300
 __writeMemory16 (C-SPY system macro). 300
 __writeMemory32 (C-SPY system macro). 301

Z

zone
 defined in device description file 135
 in C-SPY 134
 part of an absolute address. 130
 Zoom (Timeline window context menu). . 183, 186, 225, 238

Symbols

__abortLaunch (C-SPY system macro)	267
__cancelAllInterrupts (C-SPY system macro)	268
__cancelInterrupt (C-SPY system macro)	268
__clearBreak (C-SPY system macro)	269
__closeFile (C-SPY system macro)	269
__delay (C-SPY system macro)	269
__disableInterrupts (C-SPY system macro)	270
__driverType (C-SPY system macro)	270
__enableInterrupts (C-SPY system macro)	271
__evaluate (C-SPY system macro)	271
__fillMemory8 (C-SPY system macro)	272
__fillMemory16 (C-SPY system macro)	272
__fillMemory32 (C-SPY system macro)	273
__fmessage (C-SPY macro keyword)	261
__getCycleCounter (C-SPY system macro)	274
__isBatchMode (C-SPY system macro)	275
__loadImage (C-SPY system macro)	275
__memoryRestore (C-SPY system macro)	276
__memoryRestoreFromFile (C-SPY system macro)	277
__memorySave (C-SPY system macro)	277
__memorySaveToFile (C-SPY system macro)	278
__message (C-SPY macro keyword)	261
__messageBoxYesCancel (C-SPY system macro)	279
__messageBoxYesNo (C-SPY system macro)	279
__openFile (C-SPY system macro)	280
__orderInterrupt (C-SPY system macro)	281
__param (C-SPY macro keyword)	259
__readFile (C-SPY system macro)	282
__readFileByte (C-SPY system macro)	283
__readMemoryByte (C-SPY system macro)	283
__readMemory8 (C-SPY system macro)	283
__readMemory16 (C-SPY system macro)	284
__readMemory32 (C-SPY system macro)	284
__registerMacroFile (C-SPY system macro)	285
__resetFile (C-SPY system macro)	285
__setCodeBreak (C-SPY system macro)	286
__setComplexBreak (C-SPY system macro)	287
__setDataBreak (C-SPY system macro)	289
__setLogBreak (C-SPY system macro)	291
__setSimBreak (C-SPY system macro)	292
__setTraceStartBreak (C-SPY system macro)	293
__setTraceStopBreak (C-SPY system macro)	294
__smessage (C-SPY macro keyword)	261
__sourcePosition (C-SPY system macro)	295
__strFind (C-SPY system macro)	295
__subString (C-SPY system macro)	296
__targetDebuggerVersion (C-SPY system macro)	296
__toLowerCase (C-SPY system macro)	297
__toString (C-SPY system macro)	297
__toUpperCase (C-SPY system macro)	298
__unloadImage (C-SPY system macro)	298
__var (C-SPY macro keyword)	258
__writeFile (C-SPY system macro)	299
__writeFileByte (C-SPY system macro)	299
__writeMemoryByte (C-SPY system macro)	300
__writeMemory8 (C-SPY system macro)	300
__writeMemory16 (C-SPY system macro)	300
__writeMemory32 (C-SPY system macro)	301
-f (cspybat option)	326
-p (C-SPY command line option)	331
-v (C-SPY command line option)	333
--attach_to_running_target (C-SPY command line option)	315
--avrone_jtag_clock (C-SPY command line option)	316
--backend (C-SPY command line option)	316
--code_coverage_file (C-SPY command line option)	317
--cpu (C-SPY command line option)	317
--cycles (C-SPY command line option)	318
--debugfile (cspybat option)	318
--disable_internal_eeeprom (C-SPY command line option)	319
--disable_interrupts (C-SPY command line option)	319
--download_only (C-SPY command line option)	319
--drv_communication (C-SPY command line option)	319
--drv_communication_log (C-SPY command line option)	320
--drv_debug_port (C-SPY command line option)	320
--drv_download_data (C-SPY command line option)	321

--drv_dragon (C-SPY command line option) 321
 --drv_power_debugger (C-SPY command line option) . . . 322
 --drv_preserve_app_section (C-SPY command
 line option) 322
 --drv_preserve_boot_section (C-SPY command
 line option) 322
 --drv_set_exit_breakpoint (C-SPY command
 line option) 323
 --drv_set_getchar_breakpoint
 (C-SPY command line option) 323
 --drv_set_putchar_breakpoint
 (C-SPY command line option) 324
 --drv_suppress_download (C-SPY command
 line option) 324
 --drv_use_PDI (C-SPY command line option) 325
 --drv_verify_download (C-SPY command line option) . . 325
 --eeprom_size (C-SPY command line option) 325
 --enhanced_core (C-SPY command line option) 326
 --function_profiling (cspsybat option) 326
 --jtagicemkII_use_software_breakpoints
 (C-SPY command line option) 329
 --jtagice_clock (C-SPY command line option) 327
 --jtagice_do_hardware_reset
 (C-SPY command line option) 327
 --jtagice_leave_timers_running
 (C-SPY command line option) 328
 --jtagice_preserve_eeprom (C-SPY command
 line option) 328
 --jtagice_restore_fuse (C-SPY command line option) . . . 328
 --leave_target_running (C-SPY command line option) . . . 329
 --macro (C-SPY command line option) 330
 --macro_param (C-SPY command line option) 330
 --plugin (C-SPY command line option) 331
 --program_fuses_after_download
 (C-SPY command line option) 332
 --silent (C-SPY command line option) 333
 --timeout (C-SPY command line option) 333
 --64bit_doubles (C-SPY command line option) 315
 --64k_flash (C-SPY command line option) 315

Numerics

1x Units (Symbolic Memory window context menu) 148
 --64bit_doubles (C-SPY command line option) 315
 --64k_flash (C-SPY command line option) 315
 8x Units (Memory window context menu) 142