

IAR Embedded Workbench[®]

C-SPY[®] Debugging Guide



COPYRIGHT NOTICE

© 2013–2015 IAR Systems AB.

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

DISCLAIMER

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

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

TRADEMARKS

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

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

Second edition: October 2015

Part number: UCSPY-2

Internal reference: M18, Hom7.2 , IJOA.

Brief contents

Tables	17
Preface	19
Part 1. Basic debugging	27
The IAR C-SPY Debugger	29
Getting started using C-SPY	37
Executing your application	53
Variables and expressions	71
Breakpoints	107
Memory and registers	131
Part 2. Analyzing your application	169
Trace	171
Profiling	193
Code coverage	205
Part 3. Advanced debugging	209
Interrupts	211
C-SPY macros	233
The C-SPY command line utility— <code>cspybat</code>	287
Flash loaders	299
Part 4. Additional reference information	305
Debugger options	307

Additional information on C-SPY drivers	313
Index	319

Contents

Tables	17
Preface	19
Who should read this guide	19
Required knowledge	19
How to use this guide	19
Some descriptions do not apply to your product	20
What this guide contains	20
Part 1. Basic debugging	20
Part 2. Analyzing your application	21
Part 3. Advanced debugging	21
Part 4. Additional reference information	21
Other documentation	21
User and reference guides	21
The online help system	22
Web sites	23
Document conventions	23
Typographic conventions	23
Naming conventions	24
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	29
RTOS awareness	31
Debugger concepts	31
C-SPY and target systems	32
The debugger	32
The target system	33
The application	33

C-SPY debugger systems	33
The ROM-monitor program	33
Third-party debuggers	33
C-SPY plugin modules	34
The IAR C-SPY Simulator	34
Getting started using C-SPY	37
Setting up C-SPY	37
Setting up for debugging	37
Executing from reset	38
Using a setup macro file	38
Selecting a device description file	39
Loading plugin modules	39
Starting C-SPY	39
Starting a debug session	40
Loading executable files built outside of the IDE	40
Starting a debug session with source files missing	40
Loading multiple images	41
Adapting for target hardware	42
Modifying a device description file	42
Initializing target hardware before C-SPY starts	43
Running example projects	43
Running an example project	44
Reference information on starting C-SPY	44
C-SPY Debugger main window	45
Images window	50
Get Alternative File dialog box	51
Executing your application	53
Introduction to application execution	53
Briefly about application execution	53
Source and disassembly mode debugging	53
Single stepping	54
Stepping speed	56
Running the application	57

Highlighting	58
Call stack information	58
Terminal input and output	59
Debug logging	59
Reference information on application execution	60
Disassembly window	60
Call Stack window	64
Terminal I/O window	66
Terminal I/O Log File dialog box	67
Debug Log window	68
Log File dialog box	69
Report Assert dialog box	70
Autostep settings dialog box	70
Variables and expressions	71
Introduction to working with variables and expressions	71
Briefly about working with variables and expressions	71
C-SPY expressions	72
Limitations on variable information	74
Working with variables and expressions	75
Using the windows related to variables and expressions	75
Viewing assembler variables	76
Getting started using data logging	77
Getting started using data sampling	77
Reference information on working with variables and expressions	78
Auto window	79
Locals window	81
Watch window	83
Live Watch window	85
Statics window	87
Quick Watch window	90
Symbols window	92
Resolve Symbol Ambiguity dialog box	94

Data Log window	95
Data Log Summary window	97
Data Sample Setup window	99
Data Sample window	100
Sampled Graphs window	102
Breakpoints	107
Introduction to setting and using breakpoints	107
Reasons for using breakpoints	107
Briefly about setting breakpoints	107
Breakpoint types	108
Breakpoint icons	110
Breakpoints in the C-SPY simulator	110
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	114
Reference information on breakpoints	116
Breakpoints window	117
Breakpoint Usage window	118
Code breakpoints dialog box	119
Log breakpoints dialog box	121
Data breakpoints dialog box	122
Data Log breakpoints dialog box	124
Immediate breakpoints dialog box	126
Enter Location dialog box	127
Resolve Source Ambiguity dialog box	128
Memory and registers	131
Introduction to monitoring memory and registers	131
Briefly about monitoring memory and registers	131

C-SPY memory zones	132
Stack display	133
Memory access checking	133
Monitoring memory and registers	134
Configuring C-SPY to match the memory of your device	134
Defining application-specific register groups	135
Reference information on memory and registers	136
Memory window	137
Memory Save dialog box	141
Memory Restore dialog box	142
Fill dialog box	143
Symbolic Memory window	144
Stack window	147
Register window	150
SFR Setup window	152
Edit SFR dialog box	155
Memory Configuration dialog box, for the C-SPY simulator	156
Edit Memory Range dialog box, for the C-SPY simulator	159
Memory Configuration dialog box, in C-SPY hardware debugger drivers	160
Edit Memory Range dialog box, for C-SPY hardware debugger drivers	163
Memory Access Setup dialog box	165
Edit Memory Access dialog box	167
Part 2. Analyzing your application	169
Trace	171
Introduction to using trace	171
Reasons for using trace	171
Briefly about trace	171
Requirements for using trace	172
Collecting and using trace data	172
Getting started with trace	172

Trace data collection using breakpoints	173
Searching in trace data	173
Browsing through trace data	174
Reference information on trace	174
Trace window	174
Function Trace window	177
Timeline window	177
Viewing Range dialog box	185
Trace Start breakpoints dialog box	186
Trace Stop breakpoints dialog box	187
Trace Expressions window	188
Find in Trace dialog box	189
Find in Trace window	190
Profiling	193
Introduction to the profiler	193
Reasons for using the profiler	193
Briefly about the profiler	193
Requirements for using the profiler	194
Using the profiler	194
Getting started using the profiler on function level	195
Analyzing the profiling data	195
Getting started using the profiler on instruction level	197
Selecting a time interval for profiling information	198
Reference information on the profiler	199
Function Profiler window	199
Code coverage	205
Introduction to code coverage	205
Reasons for using code coverage	205
Briefly about code coverage	205
Requirements and restrictions for using code coverage	205
Reference information on code coverage	205
Code Coverage window	206

Part 3. Advanced debugging	209
Interrupts	211
Introduction to interrupts	211
Briefly about interrupt logging	211
Briefly about the interrupt simulation system	212
Interrupt characteristics	213
Interrupt simulation states	213
C-SPY system macros for interrupt simulation	214
Target-adapting the interrupt simulation system	215
Using the interrupt system	215
Simulating a simple interrupt	216
Simulating an interrupt in a multi-task system	217
Getting started using interrupt logging	218
Reference information on interrupts	218
Interrupt Setup dialog box	219
Edit Interrupt dialog box	220
Forced Interrupt window	222
Interrupt Status window	223
Interrupt Log window	225
Interrupt Log Summary window	228
C-SPY macros	233
Introduction to C-SPY macros	233
Reasons for using C-SPY macros	233
Briefly about using C-SPY macros	234
Briefly about setup macro functions and files	234
Briefly about the macro language	234
Using C-SPY macros	235
Registering C-SPY macros—an overview	236
Executing C-SPY macros—an overview	236
Registering and executing using setup macros and setup files	237
Executing macros using Quick Watch	237
Executing a macro by connecting it to a breakpoint	238

Aborting a C-SPY macro	239
Reference information on the macro language	240
Macro functions	240
Macro variables	240
Macro parameters	241
Macro strings	241
Macro statements	242
Formatted output	243
Reference information on reserved setup macro function names	245
execUserPreload	245
execUserExecutionStarted	246
execUserExecutionStopped	246
execUserFlashInit	246
execUserSetup	246
execUserFlashReset	247
execUserPreReset	247
execUserReset	247
execUserExit	248
execUserFlashExit	248
Reference information on C-SPY system macros	248
__cancelAllInterrupts	250
__cancelInterrupt	251
__clearBreak	251
__closeFile	251
__delay	252
__disableInterrupts	252
__driverType	252
__enableInterrupts	253
__evaluate	253
__fillMemory8	254
__fillMemory16	255
__fillMemory32	256
__isBatchMode	257

__loadImage	257
__memoryRestore	258
__memorySave	259
__messageBoxYesCancel	260
__messageBoxYesNo	260
__openFile	261
__orderInterrupt	262
__popSimulatorInterruptExecutingStack	263
__readFile	263
__readFileByte	264
__readMemory8, __readMemoryByte	264
__readMemory16	265
__readMemory32	265
__registerMacroFile	266
__resetFile	266
__setCodeBreak	267
__setDataBreak	268
__setDataLogBreak	269
__setLogBreak	270
__setSimBreak	271
__setTraceStartBreak	272
__setTraceStopBreak	273
__sourcePosition	274
__strFind	274
__subString	275
__targetDebuggerVersion	275
__toLower	276
__toString	276
__toUpper	277
__unloadImage	277
__writeFile	278
__writeFileByte	278
__writeMemory8, __writeMemoryByte	279
__writeMemory16	279

__writeMemory32	280
Graphical environment for macros	280
Macro Registration window	281
Debugger Macros window	283
Macro Quicklaunch window	285
The C-SPY command line utility—cspybat	287
Using C-SPY in batch mode	287
Starting cspybat	287
Output	288
Invocation syntax	288
Summary of C-SPY command line options	289
General cspybat options	289
Options available for all C-SPY drivers	290
Options available for the simulator driver	290
Reference information on C-SPY command line options ...	290
--backend	290
--code_coverage_file	291
--cycles	291
--debugfile	292
--disable_interrupts	292
--download_only	292
-f	293
--flash_loader	293
--leave_running	294
--macro	294
--macro_param	294
--mapu	295
-p	295
--plugin	296
--silent	296
--timeout	296
Flash loaders	299
Introduction to the flash loader	299

Using flash loaders	299
Setting up the flash loader(s)	299
The flash loading mechanism	300
Build considerations	300
Aborting a flash loader	301
Reference information on the flash loader	301
Flash Loader Overview dialog box	301
Flash Loader Configuration dialog box	303
Part 4. Additional reference information	305
Debugger options	307
Setting debugger options	307
Reference information on general debugger options	307
Setup	308
Download	309
Images	310
Plugins	311
Extra Options	312
Additional information on C-SPY drivers	313
Reference information on C-SPY driver menus	313
<i>C-SPY driver</i>	313
Simulator menu	314
Reference information on the C-SPY simulator	316
Simulated Frequency dialog box	316
Resolving problems	316
Write failure during load	317
No contact with the target hardware	317
Slow stepping speed	318
Index	319

Tables

1: Typographic conventions used in this guide	23
2: Naming conventions used in this guide	24
3: C-SPY assembler symbols expressions	73
4: Handling name conflicts between hardware registers and assembler labels	73
5: Project options for enabling the profiler	195
6: Project options for enabling code coverage	206
7: Timer interrupt settings	217
8: Examples of C-SPY macro variables	241
9: Summary of system macros	248
10: __cancelInterrupt return values	251
11: __disableInterrupts return values	252
12: __driverType return values	253
13: __enableInterrupts return values	253
14: __evaluate return values	254
15: __isBatchMode return values	257
16: __loadImage return values	257
17: __messageBoxYesCancel return values	260
18: __messageBoxYesNo return values	260
19: __openFile return values	261
20: __readFile return values	263
21: __setCodeBreak return values	267
22: __setDataBreak return values	268
23: __setDataLogBreak return values	269
24: __setLogBreak return values	270
25: __setSimBreak return values	271
26: __setTraceStartBreak return values	272
27: __setTraceStopBreak return values	273
28: __sourcePosition return values	274
29: __unloadImage return values	278
30: cspybat parameters	288

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 CPUNAME 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 processor (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 21.

How to use this guide

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

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

If you are new to using IAR Embedded Workbench, we suggest that you first read the guide *Getting Started with IAR Embedded Workbench®* for an overview of the tools and the features that the IDE offers. The tutorials, which you can find in the IAR Information Center, will help you get started using IAR Embedded Workbench.

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

SOME DESCRIPTIONS DO NOT APPLY TO YOUR PRODUCT

This guide describes the generic parts of the IAR C-SPY debugger.

However, some functionality described does not apply to all C-SPY drivers. You can find a list of features supported by your C-SPY driver in the release notes. Reference information about windows and dialog boxes that do not apply to all C-SPY drivers have a note that says *Requirements: See the release notes.*

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 CPUNAME.

PART I. BASIC DEBUGGING

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

PART 2. ANALYZING YOUR APPLICATION

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

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.
- *Using C-SPY macros* describes the C-SPY macro system, its features, the purposes of these features, and how to use them.
- *The C-SPY command line utility—cspybat* describes how to use C-SPY in batch mode.
- *Flash loaders* describes the flash loader, what it is and how to use it.

PART 4. ADDITIONAL REFERENCE INFORMATION

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

Other documentation

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

USER AND REFERENCE GUIDES

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

- System requirements and information about how to install and register the IAR Systems products, is available in the booklet *Quick Reference* (available in the product box) and the *Installation and Licensing Guide*.

- Getting started using IAR Embedded Workbench and the tools it provides, is available in the guide *Getting Started with IAR Embedded Workbench®*.
- Using the IDE for project management and building, is available in the *IDE Project Management and Building Guide*.
- Using the IAR C-SPY® Debugger, is available in the *C-SPY® Debugging Guide*.
- Programming for the IAR C/C++ Compiler, is available in the *IAR C/C++ Compiler Reference Guide* if your product package includes the IAR XLINK Linker, or in the *IAR C/C++ Development Guide, Compiling and Linking* if your product package includes the IAR ILINK Linker.
- 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, is available in the *IAR Assembler Reference Guide*.
- Using the IAR DLIB Library, is available in the *DLIB Library Reference information*, available in the online help system.
- Using the IAR CLIB Library, is available in the *IAR C Library Functions Reference Guide*, available in the online help system. This guide is only available if your product package includes the CLIB library.
- 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*.

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 library, you will get reference information for the DLIB library.

WEB SITES

Recommended web sites:

- The Chip manufacturer's web site.
- The IAR Systems web site, www.iar.com, that holds application notes and other product information.
- The web site of the C standardization working group, www.open-std.org/jtc1/sc22/wg14.
- The web site of the C++ Standards Committee, www.open-std.org/jtc1/sc22/wg21.
- Finally, the Embedded C++ Technical Committee web site, www.caravan.net/ec2plus, that contains information about the Embedded C++ standard.

Document conventions

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

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

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:

Style	Used for
<code>computer</code>	<ul style="list-style-type: none"> • Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
<i>parameter</i>	A placeholder for an actual value used as a parameter, for example <i>filename.h</i> where <i>filename</i> represents the name of the file. Note that this style is also used for <i>cpuname</i> , <i>configfile</i> , <i>libraryfile</i> , and other labels representing your product, as well as for the numeric part of filename extensions— <i>xx</i> .
[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.

Table 1: Typographic conventions used in this guide





Style	Used for
[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 (Continued)

NAMING CONVENTIONS

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

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

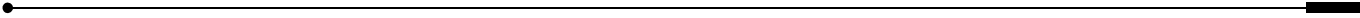
Table 2: Naming conventions used in this guide

Note that some of these products and tools might not be available in the product package you are using.

Part I. Basic debugging

This part of the *C-SPY® Debugging Guide* 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
- The IAR C-SPY Simulator

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.

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.

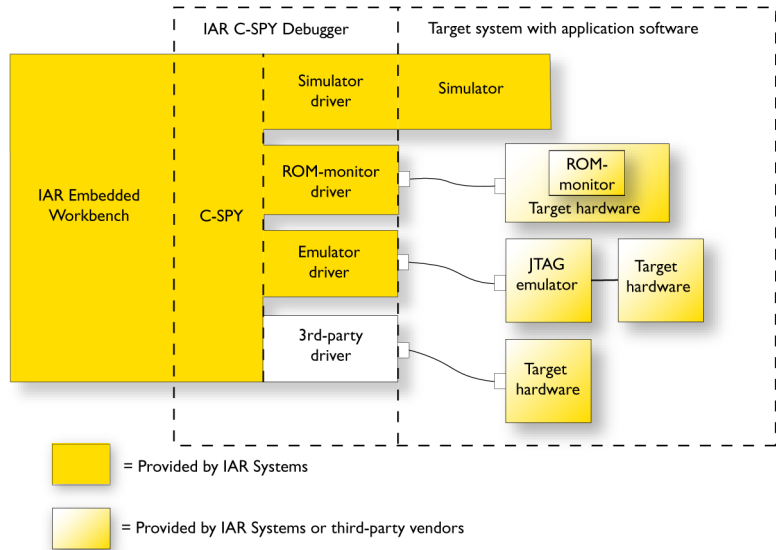
These topics are covered:

- 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.

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. Or if you are using the ILINK linker, any of the formats ELF/DWARF, Intel-extended, or Motorola. For information about which format to use with a third-party debugger, see the user documentation supplied with that tool.

C-SPY PLUGIN MODULES

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

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

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

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

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.

Getting started using C-SPY

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

Setting up C-SPY

These tasks are covered:

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

SETTING UP FOR DEBUGGING

- 1 Install a USB driver or some other communication driver if your C-SPY driver requires it.
- 2 Before you start C-SPY, choose **Project>Options>Debugger>Setup** and select the C-SPY driver that matches your debugger system: simulator or a hardware debugger system.
- 3 In the **Category** list, select the appropriate C-SPY driver and make your settings.
For information about these options, see *Debugger options*, page 305.
- 4 Click **OK**.
- 5 Choose **Tools>Options** to open the **IDE Options** dialog box:
 - Select **Debugger** to configure the debugger behavior
 - Select **Stack** to configure the debugger's tracking of stack usage.

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

See also *Adapting for target hardware*, page 42.

EXECUTING FROM RESET

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

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

If you leave the check box empty, the program counter will contain the regular hardware reset address at each reset

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 233. For an example of how to use a setup macro file, see *Initializing target hardware before C-SPY starts*, page 43.

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

For more information about device description files, see *Adapting for target hardware*, page 42.

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 309.

Starting C-SPY

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

These tasks are covered:

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

STARTING A DEBUG SESSION

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



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




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

LOADING EXECUTABLE FILES BUILT OUTSIDE OF THE IDE

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

To create a project for an externally built file:

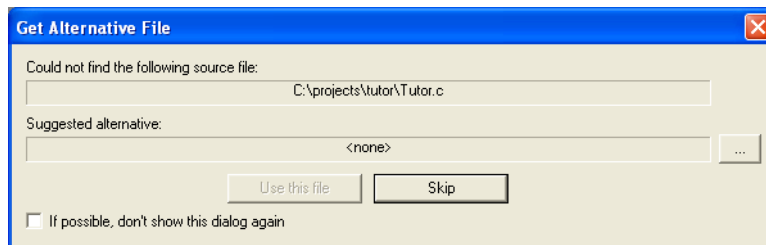
- 1 Choose **Project>Create New Project**, and specify a project name.
- 2 To add the executable file to the project, choose **Project>Add Files** and make sure to choose **All Files** in the **Files of type** drop-down list. Locate the executable file.
- 3  To start the executable file, click the **Download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

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

STARTING A DEBUG SESSION WITH SOURCE FILES MISSING

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

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



Typically, you can use the dialog box like this:

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

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

For more information, see *Get Alternative File dialog box*, page 51.

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 308.
- 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 235.

To display a list of loaded images:

Choose **Images** from the **View** menu. The **Images** window is displayed, see *Images window*, page 50.

Adapting for target hardware

These topics are covered:

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

See also *Configuring C-SPY to match the memory of your device*, page 134.

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 39. They contain device-specific information such as:

- Memory information for device-specific memory zones, see *C-SPY memory zones*, page 132. If you are using a C-SPY hardware debugger driver, the memory information retrieved from the device description file is not always sufficient, see *Memory Configuration dialog box, in C-SPY hardware debugger drivers*, page 160.
- 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 211.

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 39.

INITIALIZING TARGET HARDWARE BEFORE C-SPY STARTS

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

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

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

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

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

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

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

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

Running example projects

These tasks are covered:

- Running an example project

RUNNING AN EXAMPLE PROJECT

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

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

To run an example project:

- 1 Choose **Help>Information Center** and click **EXAMPLE PROJECTS**.
- 2 In the dialog box that appears, choose a destination folder for your project.
- 3 The available example project(s) are displayed in the workspace window. Select one of the projects, and if it is not the active project (highlighted in bold), right-click it and choose **Set As Active** from the context menu.
- 4 To view the project settings, select the project and choose **Options** from the context menu. Verify the settings for device selection and **Debugger>Setup>Driver**. As for other settings, the project is set up to suit the target system you selected.

For more information about the C-SPY options and how to configure C-SPY to interact with the target board, see *Debugger options*, page 305.

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



- 5 To compile and link the application, choose **Project>Make** or click the **Make** button.

- 6 To start C-SPY, choose **Project>Debug** or click the **Download and Debug** button. If C-SPY fails to establish contact with the target system, see *Resolving problems*, page 314.



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

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

Reference information on starting C-SPY

Reference information about:

- *C-SPY Debugger main window*, page 45
- *Images window*, page 50
- *Get Alternative File dialog box*, page 51

See also:

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

C-SPY Debugger main window

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

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

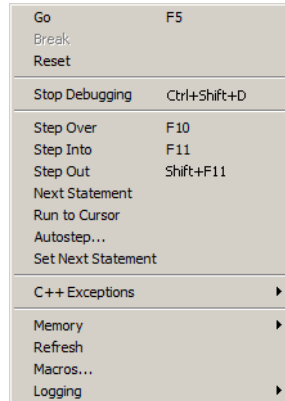
Provides commands for executing and debugging the source application. Most of the commands are also available as icon buttons on the debug toolbar.

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 311.

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:



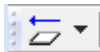
Go F5

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 autostepping, see *Autostep settings dialog box*, page 70.

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 141.

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 142.

Refresh

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

Macros

Displays a dialog box where you can list, register, and edit your macro files and functions, see *Using C-SPY macros*, page 235.

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 69.

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 67

C-SPY windows

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

- C-SPY Debugger main window
- Disassembly window
- Memory window
- Symbolic Memory window
- Register window
- Watch window
- Locals window
- Auto window
- Live Watch window
- Quick Watch window
- Statics window
- Call Stack window
- Trace window
- Function Trace window

- Timeline window
- Terminal I/O window
- Code Coverage window
- Function Profiler window
- Images window
- Stack window
- Symbols window.

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

Editing in C-SPY windows

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

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

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

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

```
myArray;3
```

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

```
myPtr;3
```

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

```
myArray;5,10
```

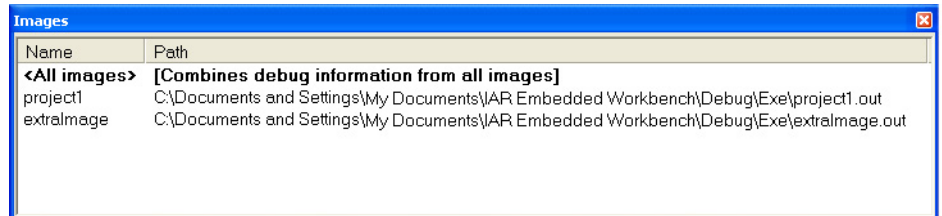
To display `myPtr+10`, `myPtr+11`, `myPtr+12`, `myPtr+13`, and `myPtr+14`, write:

```
myPtr;5,10
```

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

Images window

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



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

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

Requirements

None; this window is always available.

Display area

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

This area lists the loaded images in these columns:

Name

The name of the loaded image.

Path

The path to the loaded image.

Context menu

This context menu is available:



These commands are available:

Show all images

Shows debug information for all loaded debug images.

Show only *image*

Shows debug information for the selected debug image.

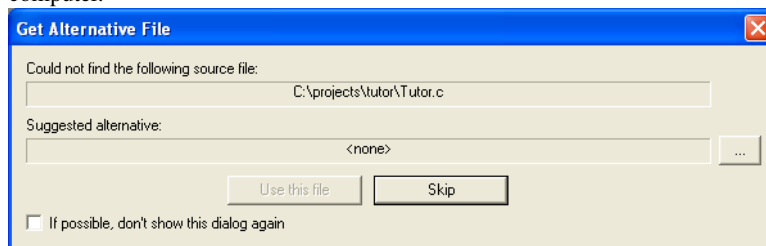
Related information

For related information, see:

- *Loading multiple images*, page 41
- *Images*, page 308
- *__loadImage*, page 257.

Get Alternative File dialog box

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



Could not find the following source file

The missing source file.

Suggested alternative

Specify an alternative file.

Use this file

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

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

Skip

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

If possible, don't show this dialog again

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

Related information

For related information, see *Starting a debug session with source files missing*, page 40.

Executing your application

- Introduction to application execution
- Reference information on application execution

Introduction to application execution

These topics are covered:

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

BRIEFLY ABOUT APPLICATION EXECUTION

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

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

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

SOURCE AND DISASSEMBLY MODE DEBUGGING

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

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

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

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

SINGLE STEPPING

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

There are several factors that can slow down the stepping speed. If you find it too slow, see *Slow stepping speed*, page 316 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 70.

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

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



Step Into

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

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

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



Step Over

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

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



Next Statement

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

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



Step Out

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

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

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

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

STEPPING SPEED

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

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

For this reason, it can be helpful to keep track of how many hardware breakpoints are used and make sure to some of them are left for stepping. For more information, see and *Breakpoint consumers*, page 110.

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

- If Trace or Function profiling is enabled. This might slow down stepping because collected Trace data is processed after each step. Note that it is not sufficient to close the corresponding windows to disable Trace data collection. Instead, you must disable the **Enable/Disable** button in both the Trace and the Function profiling windows.
- If the **Register** window is open and displays SFR registers. This might slow down stepping because all registers in the selected register group must be read from the hardware after each step. To solve this, you can choose to view only a limited selection of SFR register; you can choose between two alternatives. Either type `#SFR_name` (where `#SFR_name` reflects the name of the SFR you want to monitor) in the **Watch** window, or create your own filter for displaying a limited group of SFRs in the **Register** window. See *Defining application-specific register groups*, page 135.
- If any of the **Memory** or **Symbolic** memory windows is open. This might slow down stepping because the visible memory must be read after each step.
- If any of the expression related windows such as **Watch**, **Live Watch**, **Locals**, **Statics** is open. This might slow down stepping speed because all these windows reads memory after each step.
- If the **Stack** window is open and especially if the option **Enable graphical stack display and stack usage tracking** option is enabled. To disable this option, choose **Tools>Options>Stack** and disable it.
- If a too slow communication speed has been set up between C-SPY and the target board/emulator you should consider to increase the speed, if possible.

RUNNING THE APPLICATION



Go

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



Run to Cursor

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

HIGHLIGHTING

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

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

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

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

CALL STACK INFORMATION

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



Typically, this is useful for two purposes:

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

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

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

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

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

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

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 66 and *Terminal I/O Log File dialog box*, page 67.

DEBUG LOGGING

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



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

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

Reference information on application execution

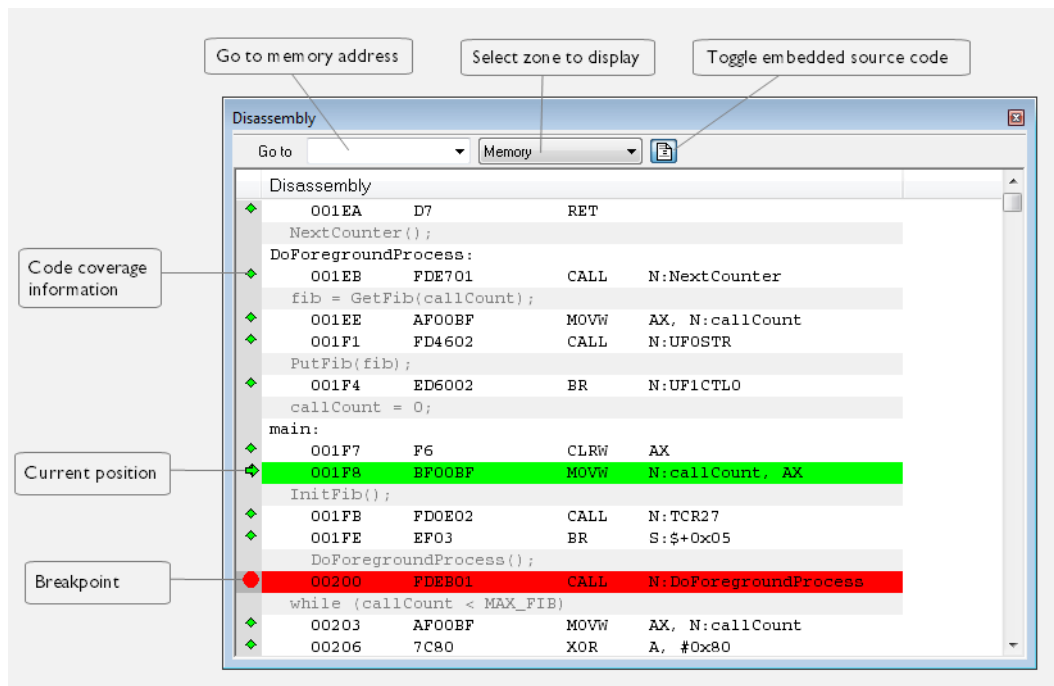
Reference information about:

- *Disassembly window*, page 60
- *Call Stack window*, page 64
- *Terminal I/O window*, page 66
- *Terminal I/O Log File dialog box*, page 67
- *Debug Log window*, page 68
- *Log File dialog box*, page 69
- *Report Assert dialog box*, page 70
- *Autostep settings dialog box*, page 70

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

Disassembly window

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



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

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

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



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

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

Go to

The memory location or symbol you want to view.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 132.

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.

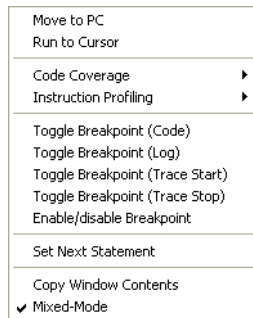
This area contains these graphic elements:

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 107.
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 119.

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 186.

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 187.

Enable/Disable Breakpoint

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

Edit Breakpoint

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

Set Next Statement

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

Copy Window Contents

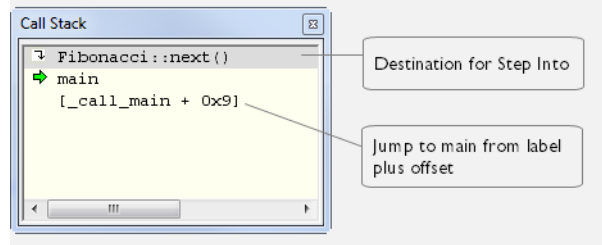
Copies the selected contents of the **Disassembly** window to the clipboard.

Mixed-Mode

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

Call Stack window

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



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

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

Requirements

None; this window is always available.

Display area

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

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

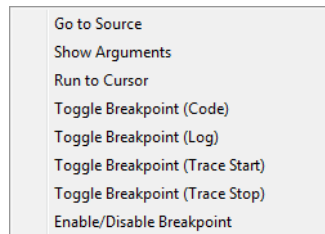
where

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

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

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)

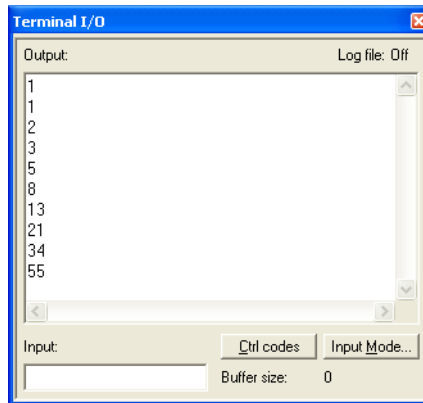
Toggles a Trace Stop breakpoint. When the breakpoint is triggered, trace data collection stops. Note that this menu command is only available if the C-SPY driver you are using supports it.

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** (for the XLINK linker) and **Include C-SPY debugging support** (for the ILINK linker).

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.

Requirements

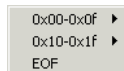
None; this window is always available.

Input

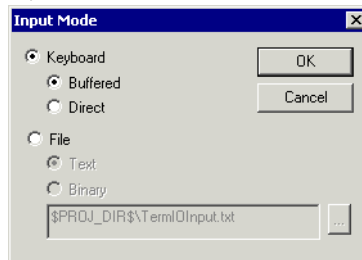
Type the text that you want to input to your application.

Ctrl codes

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

**Input Mode**

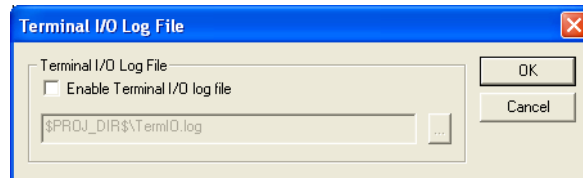
Opens the **Input Mode** dialog box where you choose whether to input data from the keyboard or from a file.



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

Terminal I/O Log File dialog box

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



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

Requirements

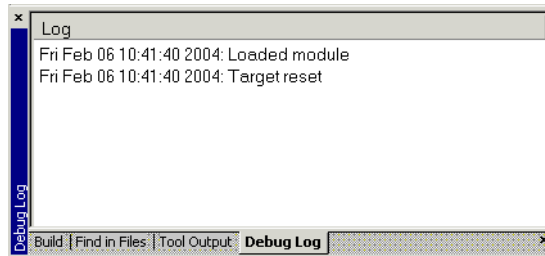
None; this dialog box is always available.

Terminal IO Log Files

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

Debug Log window

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



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

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

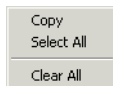
```
<path> (<row>) :<message>
<path> (<row>, <column>) :<message>
```

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

Copy

Copies the contents of the window.

Select All

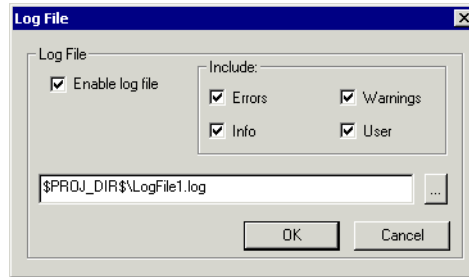
Selects the contents of the window.

Clear All

Clears the contents of the window.

Log File dialog box

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



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

Requirements

None; this dialog box is always available.

Enable Log file

Enables or disables logging to the file.

Include

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

Errors

C-SPY has failed to perform an operation.

Warnings

An error or omission of concern.

Info

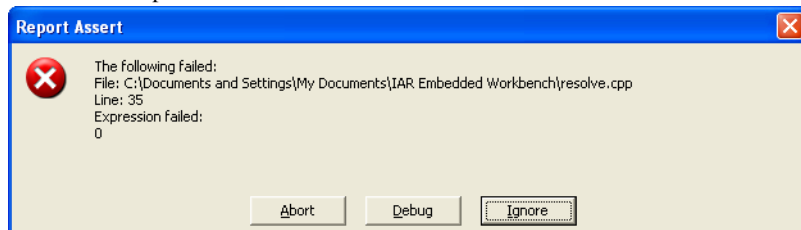
Progress information about actions C-SPY has performed.

User

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

Report Assert dialog box

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



Abort

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

Debug

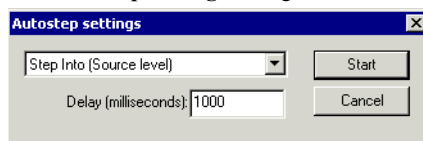
C-SPY stops the execution of the application and returns control to you.

Ignore

The assertion is ignored and the application continues to execute.

Autostep settings dialog box

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



Use this dialog box to customize autostepping.

The drop-down menu lists the available step commands.

Requirements

None; this dialog box is always available.

Delay

Specify the delay between each step in milliseconds.

Variables and expressions

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

Introduction to working with variables and expressions

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

These topics are covered:

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

BRIEFLY ABOUT WORKING WITH VARIABLES AND EXPRESSIONS

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

- **Tooltip watch**—in the editor window—provides the simplest way of viewing the value of a variable or more complex expressions. Just point at the variable with the mouse pointer. The value is displayed next to the variable.
- The **Auto** window displays a useful selection of variables and expressions in, or near, the current statement. The window is automatically updated when execution stops.
- The **Locals** window displays the local variables, that is, auto variables and function parameters for the active function. The window is automatically updated when execution stops.
- The **Watch** window allows you to monitor the values of C-SPY expressions and variables. The window is automatically updated when execution stops.
- The **Live Watch** window repeatedly samples and displays the values of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.
- The **Statics** window displays the values of variables with static storage duration. The window is automatically updated when execution stops.

- The **Macro Quicklaunch** window and the **Quick Watch** window give you precise control over when to evaluate an expression.
- The **Symbols** window displays all symbols with a static location, that is, C/C++ functions, assembler labels, and variables with static storage duration, including symbols from the runtime library.
- The **Data Log** window and the **Data Log Summary** window display logs of accesses to up to four different memory locations you choose by setting data log breakpoints. Data logging can help you locate frequently accessed data. You can then consider whether you should place that data in more efficient memory.
- The **Data Sample** window displays samples for up to four different variables. You can also display the data samples as graphs in the **Sampled Graphs** window. By using data sampling, you will get an indication of the data value over a length of time. Because it is a sampled value, data sampling is best suited for slow-changing data.
- The Trace-related windows let you inspect the program flow up to a specific state. For more information, see *Trace*, page 171.

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 42.

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

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

Table 3: C-SPY assembler symbols expressions

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

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

Table 4: Handling name conflicts between hardware registers and assembler labels

Which processor-specific symbols are available by default can be seen in the **Register** window, using the CPU Registers register group. See *Register window*, page 150.

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 234.

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 240.

Using sizeof

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

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

The former is for types and the latter for expressions.

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

LIMITATIONS ON VARIABLE INFORMATION

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

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

Effects of optimizations

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

Consider this example:

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

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

```
Unavailable
```

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

Working with variables and expressions

These tasks are covered:

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

USING THE WINDOWS RELATED TO VARIABLES AND EXPRESSIONS

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

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



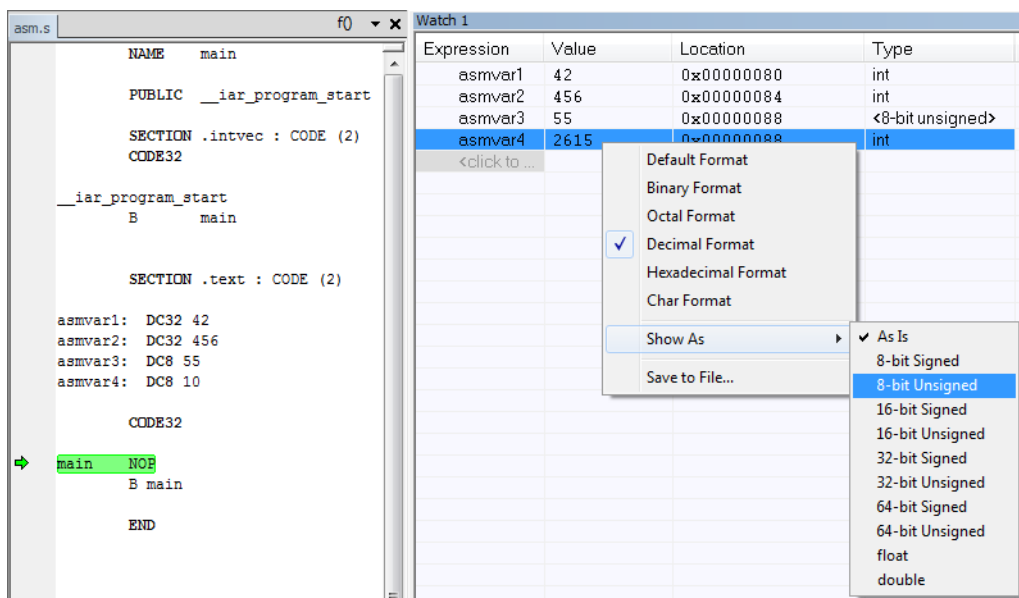
For text that is too wide to fit in a column—in any of these windows, except the **Trace** window—and thus is truncated, just point at the text with the mouse pointer and tooltip information is displayed.

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

VIEWING ASSEMBLER VARIABLES

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

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



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

GETTING STARTED USING DATA LOGGING

- 1 To set 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 109.

- 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 any of the **Data Log** window, **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.

GETTING STARTED USING DATA SAMPLING

- 1 Choose **C-SPY driver>Data Sample Setup** to open the **Data Sample Setup** window.
- 2 In the **Data Sample Setup** window, perform these actions:
 - In the **Expression** column, type the name of the variable for which you want to sample data. The variable must be an integral type with a maximum size of 32 bits and you can specify up to four variables. Make sure that the checkbox is selected for the variable that you want to sample data.

- In the **Sampling interval** column, type the number of milliseconds to pass between the samples.
- 3 To view the result of data sampling, you must enable it in the window in question:
 - Choose **C-SPY driver>Data Sample** to open the **Data Sample** window. From the context menu, choose **Enable**.
 - Choose **C-SPY driver>Sampled Graph** to open the **Sampled Graph** window. From the context menu, choose **Enable**.
 - 4 Start executing your application program. This starts the data sampling. When the execution stops, for example because a breakpoint is triggered, you can view the result either in the **Data Sample** window or as the Data Sample graph in the **Sampled Graphs** window
 - 5 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
 - 6 To disable data sampling, choose **Disable** from the context menu in each window where you have enabled it.

Reference information on working with variables and expressions

Reference information about:

- *Auto window*, page 79
- *Locals window*, page 81
- *Watch window*, page 83
- *Live Watch window*, page 85
- *Statics window*, page 87
- *Quick Watch window*, page 90
- *Symbols window*, page 92
- *Resolve Symbol Ambiguity dialog box*, page 94
- *Data Log window*, page 95
- *Data Log Summary window*, page 97
- *Data Sample Setup window*, page 99
- *Data Sample window*, page 100
- *Sampled Graphs window*, page 102

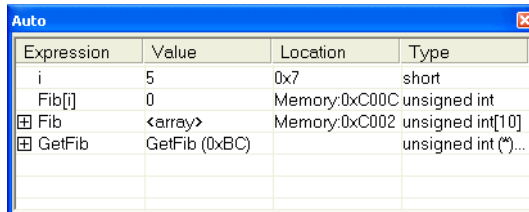
See also:

- *Reference information on trace*, page 174 for trace-related reference information

- *Macro Quicklaunch window*, page 285

Auto window

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



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

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

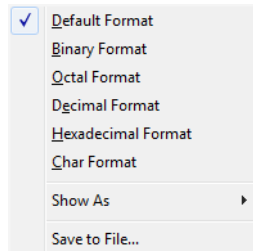
For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 76.

Options

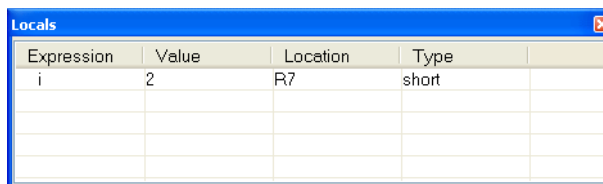
Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context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

Save to File

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

Locals window

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



Expression	Value	Location	Type
i	2	R7	short

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

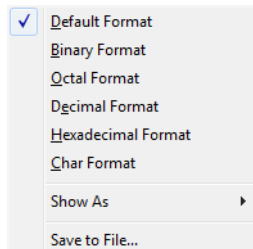
For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 76.

Options

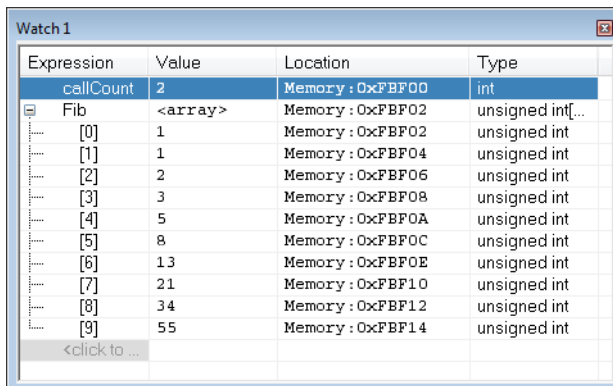
Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context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

Save to File

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

Watch window

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



Expression	Value	Location	Type
callCount	2	Memory : 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.

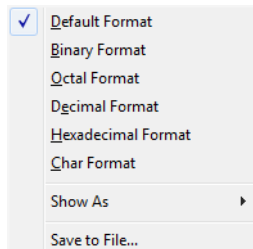
For more information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 76.

Options

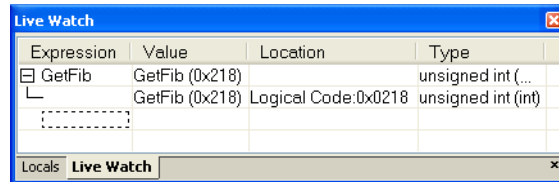
Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context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

Save to File

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

Live Watch window

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



This window repeatedly samples and displays the value of expressions while your application is executing. Variables in the expressions must be statically located, such as global variables.

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Display area

This area contains these columns:

Expression

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

Value

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

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

This column is editable.

Location

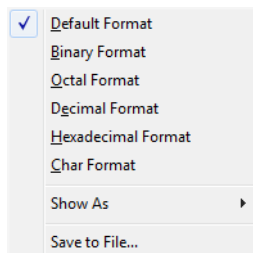
The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 76.

Options

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

Save to File

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

Statics window

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

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

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

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

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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

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

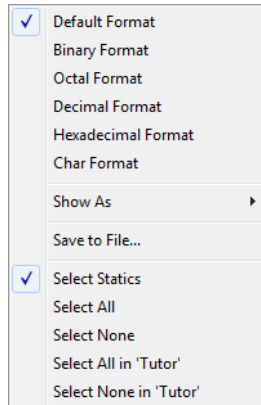
The location in memory where this variable is stored.

Type

The data type of the variable.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

Save to File

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

Select Statics

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

Select All

Selects all variables.

Select None

Deselects all variables.

Select All in *module*

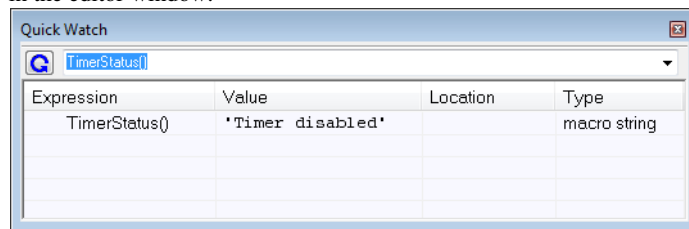
Selects all variables in the selected module.

Select None in *module*

Deselects all variables in the selected module.

Quick Watch window

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



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

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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

To evaluate an expression:

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

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

Alternatively:

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



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

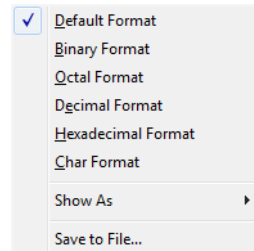
For an example, see *Using C-SPY macros*, page 235.

Requirements

None; this window is always available.

Context menu

This context menu is available:



These commands are available:

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

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

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

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

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

Show As

Displays a submenu that provides commands for changing the default type interpretation of variables. The commands on this submenu are mainly useful for assembler variables—data at assembler labels—because these are, by default, displayed as integers. For more information, see *Viewing assembler variables*, page 76.

Options

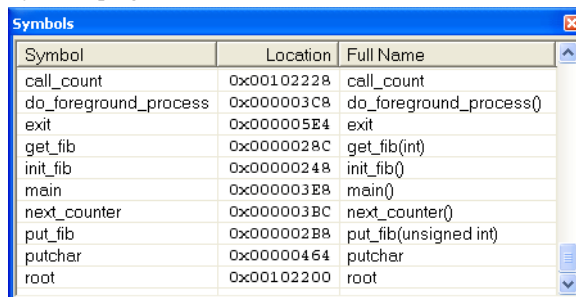
Displays the **IDE Options** dialog box where you can set the **Update interval** option. The default value of this option is 1000 milliseconds, which means the **Live Watch** window will be updated once every second during program execution. Note that this command is only available from this context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

Save to File

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

Symbols window

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



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

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

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

Requirements

None; this window is always available.

Display area

This area contains these columns:

Symbol

The symbol name.

Location

The memory address.

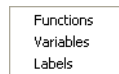
Full name

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

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

Context menu

This context menu is available:



These commands are available:

Functions

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

Variables

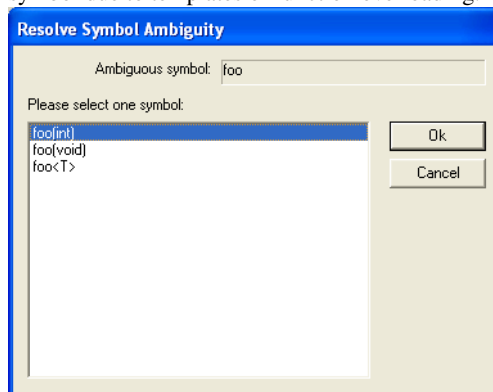
Toggles the display of variables on or off in the list.

Labels

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

Resolve Symbol Ambiguity dialog box

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



Requirements

None; this window is always available.

Ambiguous symbol

Indicates which symbol that is ambiguous.

Please select one symbol

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

Data Log window

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

Time	Program Counter	I1	Address	s2	Address
0.160us	---			W 0x0000	@ 0x2004
0.160us	0xFFE00049	-	@ 0x2000		
24.480us	0xFFE000B5			R 0x0000	@ 0x2006
24.720us	0xFFE000BF			W 0x0042	@ 0x2004
24.760us	0xFFE000C6			R 0x0042	@ 0x2006
24.960us	0xFFE000E4	W 0x00004444	@ 0x2000		
<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.

See also *Getting started using data logging*, page 77.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Display area

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

Time

The time for the data access for the C-SPY hardware debugger driver and the simulator, based on the clock frequency specified in the **Configuration** dialog box.

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

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

Cycles

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

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

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

Program Counter*

Displays one of these:

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

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

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

Value

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

To specify what data you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 109.

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

Identical to the context menu of the **Interrupt Log** window, see *Interrupt Log window*, page 225.

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 77.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Display area

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

Data

The name of the data object you have selected to log accesses to. To specify what data object you want to log accesses to, use the **Data Log** breakpoint dialog box. See *Data Log breakpoints*, page 109.

Total Accesses

The number of total accesses.

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

Read Accesses

The number of total read accesses.

Write Accesses

The number of total write accesses.

Unknown Accesses

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

Approximative time count

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

For some C-SPY drivers, this information is not displayed or the value is always zero. In this case, all logs have an exact time stamp.

For other C-SPY drivers, a non-zero value is displayed. The value represents the amount of logs with an approximative time stamp. This might happen if the bandwidth in the communication channel is too low compared to the amount of data packets generated by the CPU or if the CPU generated packets with an approximative time stamp.

Overflow count

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

For some C-SPY drivers, this information is not displayed or the value is always zero.

For other C-SPY drivers, the number represents the amount of overflows in the communication channel which can cause logs to be lost. If this happens, it indicates that logs might be incomplete. To solve this, make sure not to use all C-SPY log features simultaneously or check used bandwidth for the communication channel.

Current time|cycles

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

For some C-SPY drivers, the value is always zero or not visible at all.

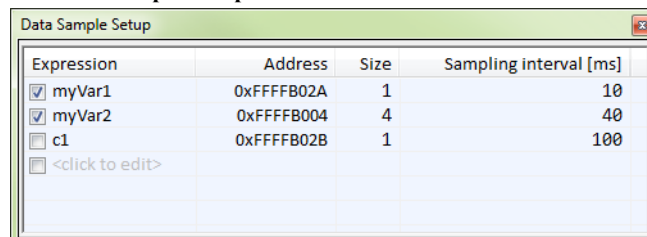
For other C-SPY drivers, the number represents the current time or cycles—the number of cycles or the execution time since the start of execution.

Context menu

Identical to the context menu of the **Interrupt Log** window, see *Interrupt Log window*, page 225.

Data Sample Setup window

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



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

See also *Getting started using data sampling*, page 77.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Display area

This area contains these columns:

Expression

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

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

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

Address

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

Size

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

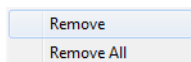
Sampling interval [ms]

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

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

Context menu

This context menu is available:



These commands are available:

Remove

Removes the selected variable.

Remove All

Removes all variables.

Data Sample window

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

The Data Sample window displays a table with the following data:

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

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

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

See also *Getting started using data sampling*, page 77.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Display area

This area contains these columns:

Sampling Time

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

The selected expression

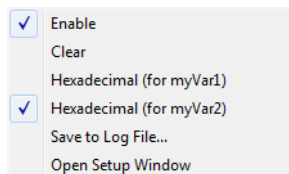
The column headers display the names of the variables that you selected in the **Data Sample Setup** window. The column cells display the sampling values for the variable.

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

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

Context menu

This context menu is available:



These commands are available:

Enable

Enables data sampling.

Clear

Clears the sampled data.

Hexadecimal (for var)

Toggles between displaying the values of selected variable in decimal or hexadecimal format. The display format affects the **Data Sample** window and the **Sampled Graphs** window.

Save to Log File

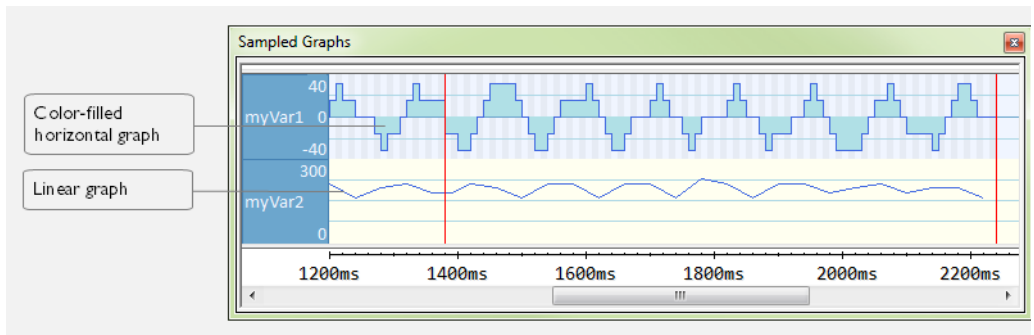
Displays a standard save dialog box.

Open setup window

Opens the **Data Sample Setup** window.

Sampled Graphs window

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



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

- The graph displays how the value of the variable changes over time. The area on the left displays the limits, or range, of the Y-axis for the variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the **Data Sample** window, see *Data Sample window*, page 100.
- The graph can be displayed as levels, where a horizontal line—optionally color-filled—shows the value until the next sample. Alternatively, the graph can be linear, where a line connects consecutive samples.
- A red vertical line indicates the time of application execution stops.

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

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

- Right-click and choose **Zoom In** or **Zoom Out** from the context menu. Alternatively, use the + and - keys to zoom.
- Right-click in the graph and choose **Navigate** and the appropriate command to move backward and forward on the graph. Alternatively, use any of the shortcut keys: arrow keys, Home, End, and Ctrl+End.

- Double-click on a sample to highlight the corresponding source code in the editor window and in the **Disassembly** window.
- Click on the graph and drag to select a time interval. Press Enter or right-click and choose **Zoom>Zoom to Selection** from the context menu. The selection zooms in.



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

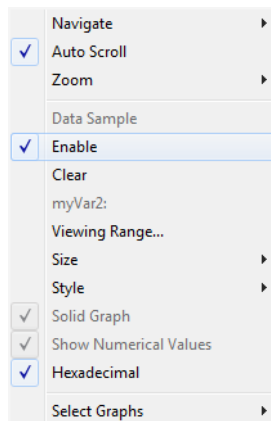
See also *Getting started using data sampling*, page 77.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Context menu

This context menu is available:



These commands are available:

Navigate

Commands for navigating in the graphs. Choose between:

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

Previous moves the selection to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll

Toggles auto scrolling on or off. When on, the most recently collected data is automatically displayed if you have executed the command **Navigate>End**.

Zoom

Commands for zooming the window, in other words, changing the time scale. Choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +.

Zoom Out zooms out on the time scale. Shortcut key: -.

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

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

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

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

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

Data Sample

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

Open Setup window (Data Sample Graph)

Opens the **Data Sample Setup** window.

Enable

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as OFF in the **Data Sample** window. If no data has been sampled for a graph, *no data* will appear instead of the graph.

Clear

Clears the sampled data.

Variable

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

Viewing Range

Displays a dialog box, see *Viewing Range dialog box*, page 185.

Size

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

Style

Choose how to display the graph. Choose between:

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

Linear, where a line connects consecutive samples.

Solid Graph

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

Hexadecimal (for var)

Toggles between displaying the selected variable in decimal or hexadecimal format. The display format affects the **Data Sample** window and the **Sampled Graphs** window.

Show Numerical Value

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

Select Graphs

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

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
- 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 54.

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 up to the maximum size that your microcontroller supports. If you specify a data log breakpoint on a memory location that cannot be accessed by one instruction, for example a `double` or a too large area in the **Memory** window, the result might not be what you intended.

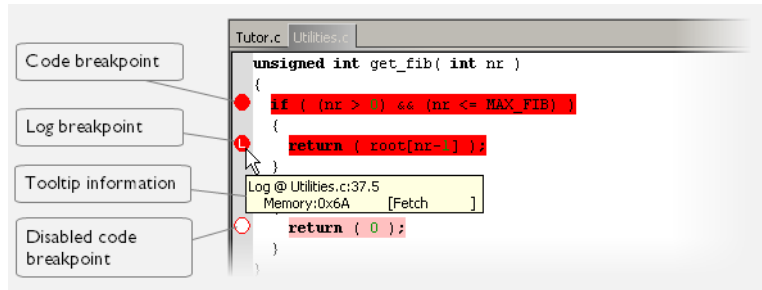
Immediate breakpoints

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

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

BREAKPOINT ICONS

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



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



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.

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** (for the XLINK linker) and **Include C-SPY debugging support** (for the ILINK linker) 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.

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.

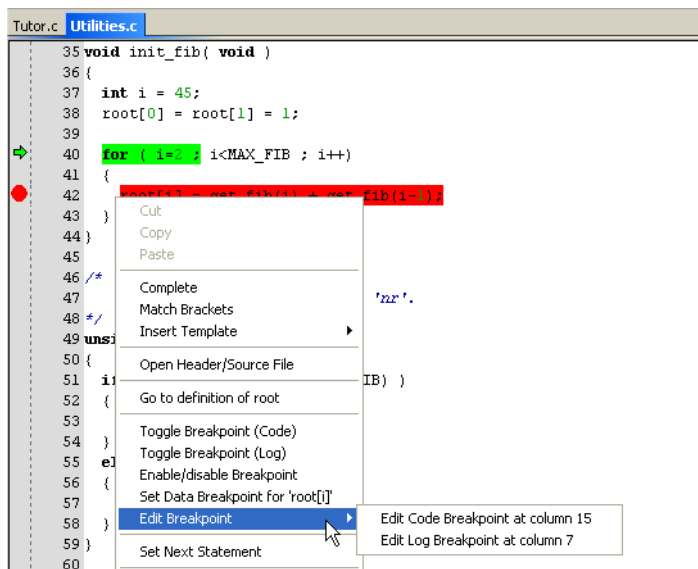
Depending on the C-SPY driver you are using, different breakpoint types are available.

- 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.

For information about each breakpoint macro, see *Reference information on C-SPY system macros*, page 248.

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 235.

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:

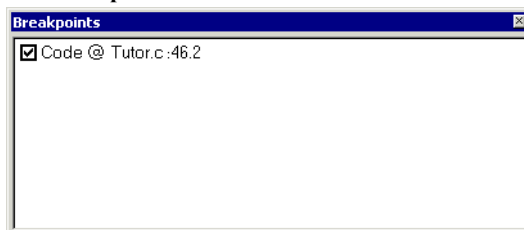
- *Breakpoints window*, page 117
- *Breakpoint Usage window*, page 118
- *Code breakpoints dialog box*, page 119
- *Log breakpoints dialog box*, page 121
- *Data breakpoints dialog box*, page 122
- *Data Log breakpoints dialog box*, page 124
- *Immediate breakpoints dialog box*, page 126
- *Enter Location dialog box*, page 127
- *Resolve Source Ambiguity dialog box*, page 128.

See also:

- *Reference information on C-SPY system macros*, page 248
- *Reference information on trace*, page 174.

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

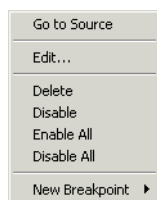
None; this window is always available.

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

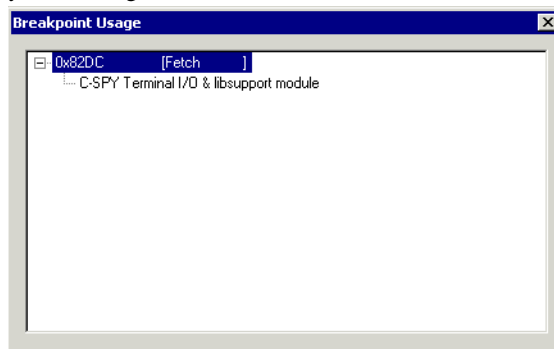
Disables all defined breakpoints.

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. If your target system has a limited number of hardware breakpoints and software breakpoints are not enabled, exceeding the number of available hardware breakpoints will cause the debugger to single step. This will significantly reduce the execution speed. Therefore, in a debugger system with a limited amount of hardware breakpoints, you can use the **Breakpoint Usage** window for:

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

Requirements

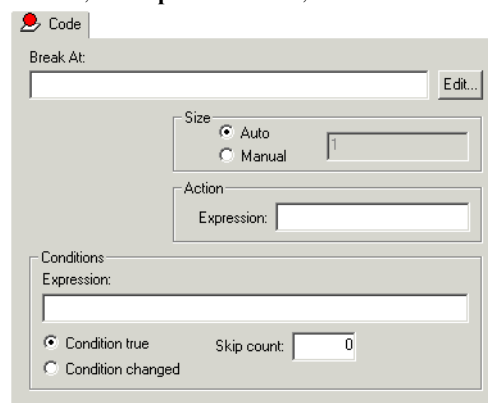
None; this window is always available.

Display area

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

Code breakpoints dialog box

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



This figure reflects the C-SPY simulator.

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

Requirements

See *Supported debugger features* in the release notes for information about applicability.

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 127.

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 114.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 72.

Condition true

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

Condition changed

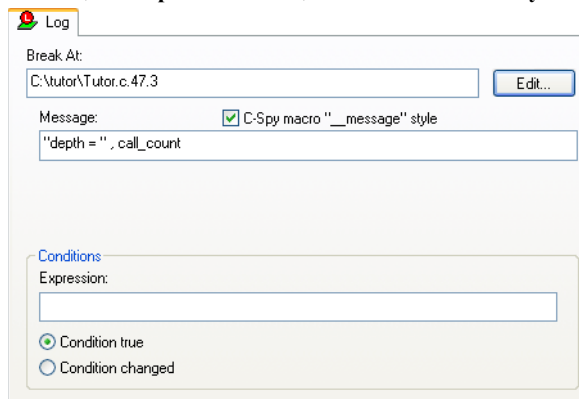
The breakpoint is triggered if the value of the expression has changed since it was last evaluated.

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.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

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 127.

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 243.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 72.

Condition true

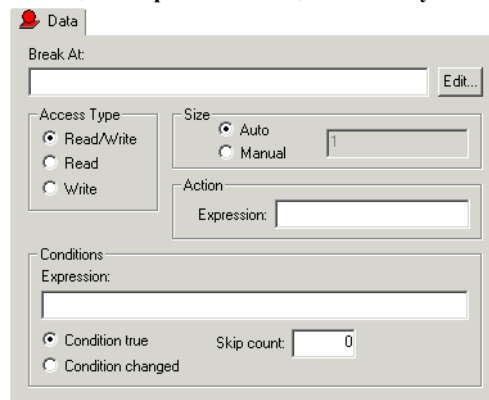
The breakpoint is triggered if the value of the expression is true.

Condition changed

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

Data breakpoints dialog box

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



This figure reflects the C-SPY simulator.

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

Requirements

See *Supported debugger features* in the release notes for information about applicability.

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 127.

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 114.

Conditions

Specify simple or complex conditions:

Expression

Specify a valid C-SPY expression, see *C-SPY expressions*, page 72.

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 also *Data Log breakpoints*, page 109 and *Getting started using data logging*, page 77.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

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 127.

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

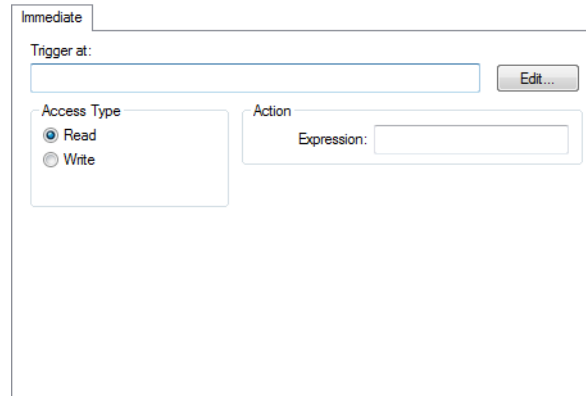
The size will automatically be based on the type of expression the breakpoint is set on. For example, if you set the breakpoint on a 12-byte structure, the size of the breakpoint will be 12 bytes.

Manual

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

Immediate breakpoints dialog box

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



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

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 127.

Requirements

The C-SPY simulator.

Access Type

Selects the type of memory access that triggers the breakpoint:

Read

Reads from location.

Write

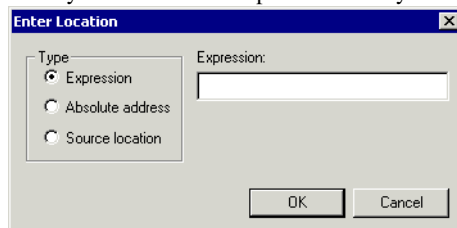
Writes to location.

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 114.

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 72.

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 132.

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.

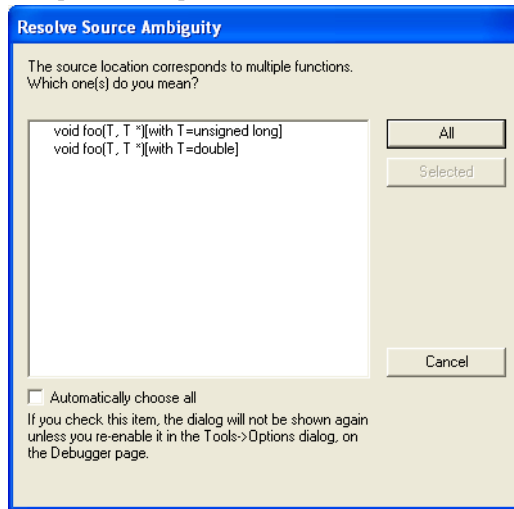
For example, `{C:\src\prog.c}.22.3`

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*.

Memory and registers

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

Introduction to monitoring memory and registers

These topics are covered:

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

BRIEFLY ABOUT MONITORING MEMORY AND REGISTERS

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

- The **Memory** window

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

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

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

- The **Register** window

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

- The **SFR Setup** window

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

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



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

C-SPY MEMORY ZONES

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

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.

For more information, see *Selecting a device description file*, page 39 and *Modifying a device description file*, page 42.

STACK DISPLAY

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

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

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

Stack usage

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



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

Note: The size and location of the stack is retrieved from the definition of the segment holding the stack, made in the linker configuration file. If you, for some reason, modify the stack initialization made in the system startup code, `cstartup`, you should also change the 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*.

MEMORY ACCESS CHECKING

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

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

Monitoring memory and registers

These tasks are covered:

- *Configuring C-SPY to match the memory of your device*, page 134.
- *Defining application-specific register groups*, page 135.

CONFIGURING C-SPY TO MATCH THE MEMORY OF YOUR DEVICE

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

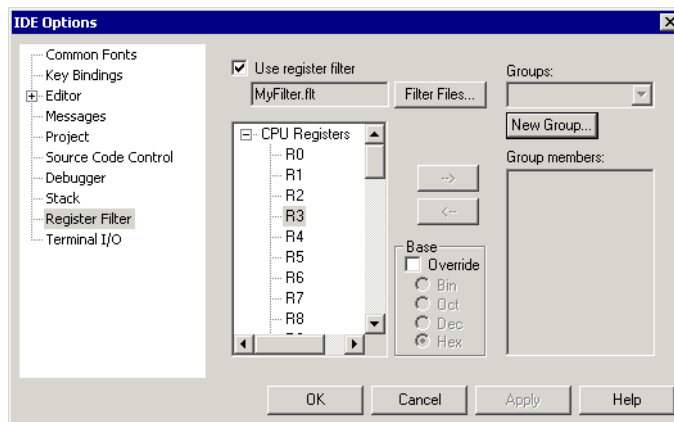
However, if that file does not specify memory ranges for a specific device, but instead for a family of devices (perhaps with varying amounts of on-chip RAM), you should tailor the areas to match your device.

To fine-tune the memory areas to suit your device, use the *Memory Configuration dialog box, for the C-SPY simulator*, page 156 and *Memory Configuration dialog box, in C-SPY hardware debugger drivers*, page 160.

DEFINING APPLICATION-SPECIFIC REGISTER GROUPS

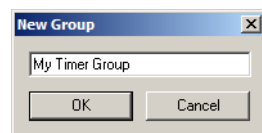
Defining application-specific register groups minimizes the amount of registers displayed in the **Register** window and speeds up the debugging.

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



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

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



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

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

Note: The registers that appear in the list of registers are retrieved from the `da.f` file that is currently used. If a certain SFR that you need does not appear, you can register your own SFRs. For more information, see *SFR Setup window*, page 152.

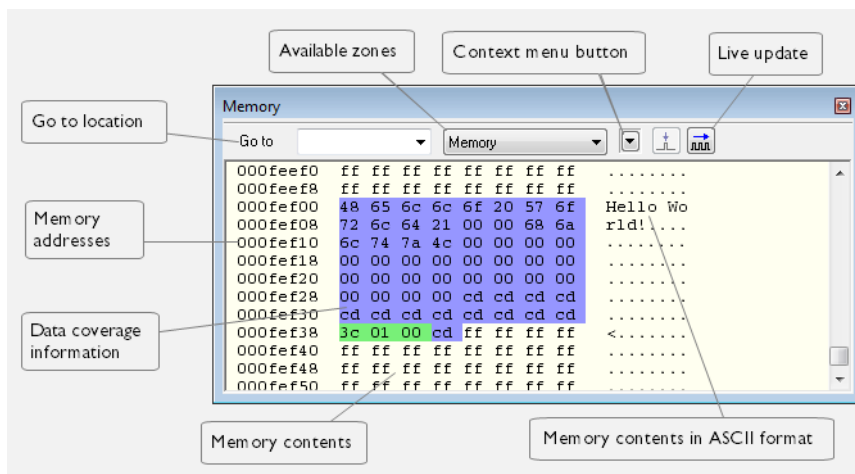
Reference information on memory and registers

Reference information about:

- *Memory window*, page 137
- *Memory Save dialog box*, page 141
- *Memory Restore dialog box*, page 142
- *Fill dialog box*, page 143
- *Symbolic Memory window*, page 144
- *Stack window*, page 147
- *Register window*, page 150
- *SFR Setup window*, page 152
- *Edit SFR dialog box*, page 155
- *Memory Configuration dialog box, for the C-SPY simulator*, page 156
- *Edit Memory Range dialog box, for the C-SPY simulator*, page 159
- *Memory Configuration dialog box, in C-SPY hardware debugger drivers*, page 160
- *Edit Memory Range dialog box, for C-SPY hardware debugger drivers*, page 163
- *Memory Access Setup dialog box*, page 165
- *Edit Memory Access dialog box*, page 167.

Memory window

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



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



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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

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 132.

Context menu button

Displays the context menu.

Update Now

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

Live Update

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

Display area

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

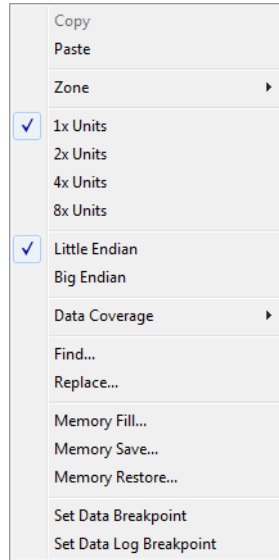
Data coverage is displayed with these colors:

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

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

Context menu

This context menu is available:



These commands are available:

Copy, Paste

Standard editing commands.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 132.

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*.

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*.

Memory Fill

Displays a dialog box, where you can fill a specified area with a value, see *Fill dialog box*, page 143.

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 141.

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 142.

Set Data Breakpoint

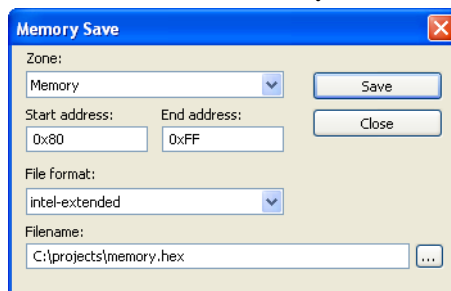
Sets breakpoints directly in the **Memory** window. The breakpoint is not highlighted; you can see, edit, and remove it in the **Breakpoints** dialog box. The breakpoints you set in this window will be triggered for both read and write access. For more information, see *Setting a data breakpoint in the 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 109 and *Getting started using data logging*, page 77.

Memory Save dialog box

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



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

Requirements

None; this dialog box is always available.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 132.

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

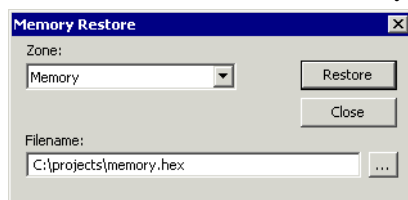
Specify the destination file to be used; a browse button is available for your convenience.

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 132.

Filename

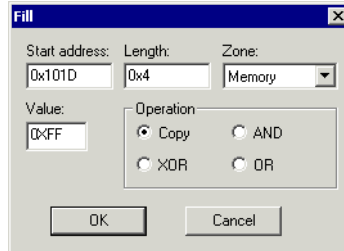
Specify the file to be read; a browse button is available for your convenience.

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 132.

Value

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

Operation

These are the available memory fill operations:

Copy

Value will be copied to the specified memory area.

AND

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

XOR

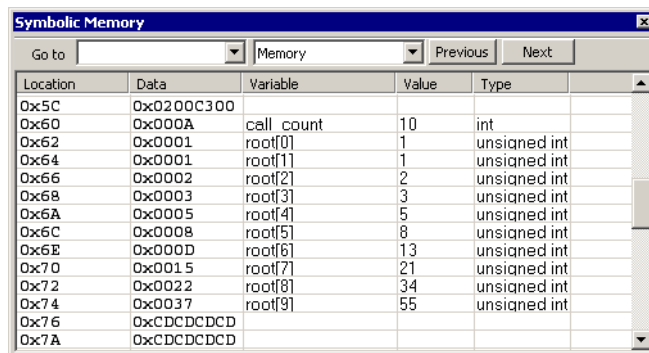
An XOR operation will be performed between Value and the existing contents of memory before writing the result to memory.

OR

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

Symbolic Memory window

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



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

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



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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

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 132.

Previous

Highlights the previous symbol in the display area.

Next

Highlights the next symbol in the display area.

Display area

This area contains these columns:

Location

The memory address.

Data

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

Variable

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

Value

The value of the variable. This column is editable.

Type

The type of the variable.

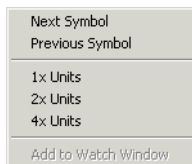
There are several different ways to navigate within the memory space:

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

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

Context menu

This context menu is available:



These commands are available:

Next Symbol

Highlights the next symbol in the display area.

Previous Symbol

Highlights the previous symbol in the display area.

1x Units

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

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

Add to Watch window

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

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

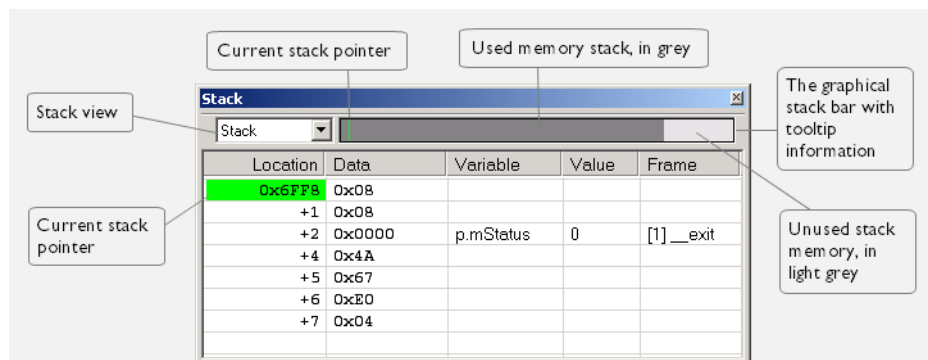
Displays the memory contents in hexadecimal format.

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. In addition, some integrity checks of the stack can be performed to detect and warn about problems with stack overflow. For example, the **Stack** window is useful for determining the optimal size of the stack.

To view the graphical stack bar:

- 1 Choose **Tools>Options>Stack**.
- 2 Select the option **Enable graphical stack display and stack usage**.

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

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

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

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

When the stack bar is enabled, the functionality needed to detect and warn about stack overflows is also enabled.



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

Display area

This area contains these columns:

Location

Displays the location in memory. The addresses are displayed in increasing order. If your target system has a stack that grows toward high addresses, the top of the stack will consequently be located at the bottom of the window. The address referenced by the stack pointer, in other words the top of the stack, is highlighted in a green color.

Data

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

Variable

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

Value

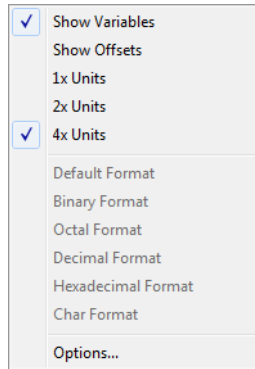
Displays the value of the variable that is displayed in the **Variable** column.

Frame

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

Context menu

This context menu is available:



These commands are available:

Show variables

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

Show offsets

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

1x Units

Displays the memory contents as single bytes.

2x Units

Displays the memory contents as 2-byte groups.

4x Units

Displays the memory contents as 4-byte groups.

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

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

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

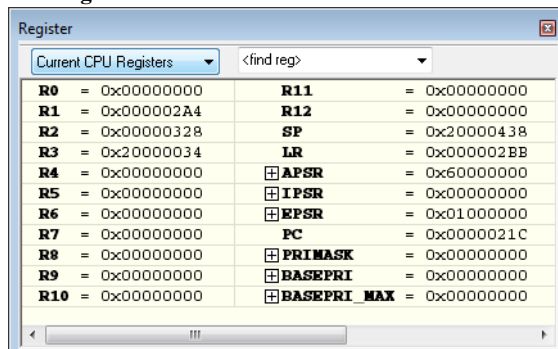
Variables	The display setting affects only the selected variable, not other variables.
Array elements	The display setting affects the complete array, that is, the same display format is used for each array element.
Structure fields	All elements with the same definition—the same field name and C declaration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

Options

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

Register window

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



This window gives an up-to-date display of the contents of the processor registers and special function registers, and allows you to edit the content of some of the registers.

Optionally, you can choose to load either predefined register groups or to define your own application-specific groups.

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

For information about editing in C-SPY windows, see *C-SPY Debugger main window*, page 45.

To enable predefined register groups:

- 1 Select a device description file that suits your device, see *Selecting a device description file*, page 39.
- 2 The register groups appear in the **Register** window, provided that they are defined in the device description file. Note that the available register groups are also listed on the **Register Filter** page.

To define application-specific register groups:

See *Defining application-specific register groups*, page 135.

Requirements

None; this window is always available.

Toolbar

The toolbar contains:

CPU Registers

Selects which register group to display, by default **CPU Registers**. Additional register groups are predefined in the device description files that make SFR registers available in the register window. The device description file contains a section that defines the special function registers and their groups. If some of your SFRs are missing, you can register your own SFRs in a Custom group, see *SFR Setup window*, page 152.

<find register>

Specify the name of a register that you want to find. Press the Enter key and the first group where this register is found is displayed. The register search box has a history depth of 20 search entries.

Display area

Displays registers and their values. Every time C-SPY stops, a value that has changed since the last stop is highlighted. Some of the registers are read-only, some of the registers are write-only (marked with *w*), and some of the registers are editable. To edit

the contents of an editable register, click it, and modify its value. Press Esc to cancel the new value.

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

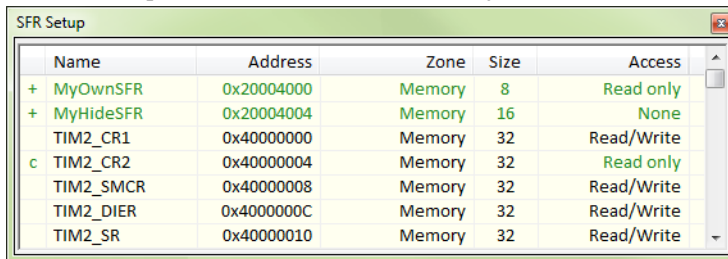
To change the display format, change the **Base** setting on the **Register Filter** page—available by choosing **Tools>Options**.

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

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

SFR Setup window

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



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

This window displays the currently defined SFRs that C-SPY has information about. You can choose to display only factory-defined or custom-defined SFRs, or both. If required, you can use this window to customize the aspects of the SFRs. For factory-defined SFRs (that is, retrieved from the `ddf` file that is currently used), you can only customize the access type.

Any custom-defined SFRs are added to a dedicated register group called Custom, which you can choose to display in the **Register** window. Your custom-defined SFRs are saved in `projectCustomSFR.sfr`.

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

Requirements

None; this window is always available.

Display area

This area contains these columns:

Status

A character that signals the status of the SFR, which can be one of:

blank, a factory-defined SFR.

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

+, a custom-defined SFR.

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

Name

A unique name of the SFR.

Address

The memory address of the SFR.

Zone

Selects a memory zone, see *C-SPY memory zones*, page 132.

Size

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

Access

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

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

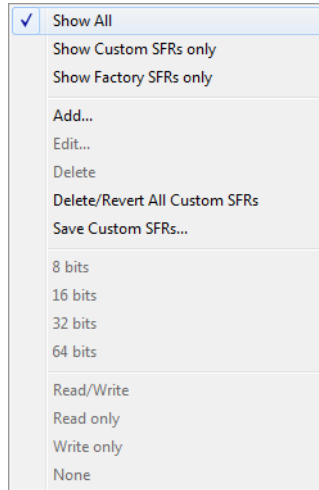
You can click a column header to sort the SFRs according to the column property.

Color coding used in the display area:

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

Context menu

This context menu is available:



These commands are available:

Show All

Shows all SFR.

Show Custom SFRs only

Shows all custom-defined SFRs.

Show Factory SFRs only

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

Add

Displays the **Edit SFR** dialog box where you can add a new SFR, see *Edit SFR dialog box*, page 155.

Edit

Displays the **Edit SFR** dialog box where you can edit an SFR, see *Edit SFR dialog box*, page 155.

Delete

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

Delete/revert All Custom SFRs

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

Save Custom SFRs

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

8|16|32|64 bits

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

Read/Write|Read only|Write only|None

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

Edit SFR dialog box

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

Use this dialog box to define the SFRs.

Requirements

None; this dialog box is always available.

Name

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

Address

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

Zone

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

Size

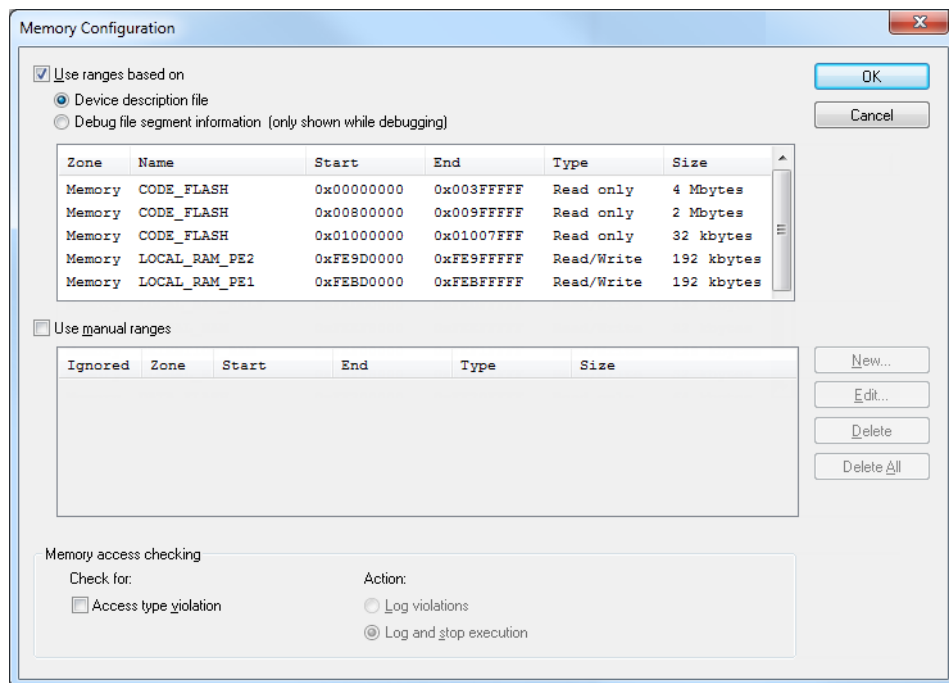
Selects the size of the SFR. Choose between **8**, **16**, **32**, or **64** bits. Note that the display format can only be changed for custom-defined SFRs.

Access

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

Memory Configuration dialog box, for the C-SPY simulator

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



To handle memory as efficiently as possible during debugging, C-SPY needs information about the memory configuration. By default, C-SPY uses a default

configuration based on information retrieved from the device description file that you have selected.

If that file does not specify memory ranges for the specific device that you are using, but instead for a family of devices (perhaps with varying amounts of on-chip RAM), use this dialog box to add memory areas so that they match the memory available on your device.

By default, the simulator does not allow accesses outside of the defined memory areas. In addition, you can use this dialog box to make C-SPY check that the access types are correct, which means that C-SPY will warn if there is a write access to read-only memory.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Use ranges based on

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

Device description file

Retrieves the memory configuration from the device description file that you have specified. See *Selecting a device description file*, page 39.

This option is used by default.

Debug file segment information

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

Memory information is displayed in these columns:

Zone

The memory zone, see *C-SPY memory zones*, page 132.

Name

The name of the memory area.

Start

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

End

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

Type

The access type of the memory area.

Size

The size of the memory area.

Use manual ranges

Specify your own ranges manually via the **Edit Memory Range** dialog box. To open this dialog box, click **New** to specify a new memory range, or select a memory range and click **Edit** to modify it. For more information, see *Edit Memory Range dialog box, for the C-SPY simulator*, page 159.

The ranges you define manually are saved between debug sessions.

An **X** in the column **Ignored** means that the specified manual range is illegal, for example because it overlaps another range. Such an area will not be used.

Memory access checking

Check for determines what to check for:

- Access type violation.

Action selects the action to be performed if an access violation occurs; choose between:

- Log violations
- Log and stop execution.

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

Buttons

These buttons are available for the manual ranges:

New

Opens the **Edit Memory Range** dialog box, where you can specify a new memory range and associate an access type with it, see *Edit Memory Range dialog box, for the C-SPY simulator*, page 159.

Edit

Opens the **Edit Memory Range** dialog box, where you can edit the selected memory area. See *Edit Memory Range dialog box, for the C-SPY simulator*, page 159.

Delete

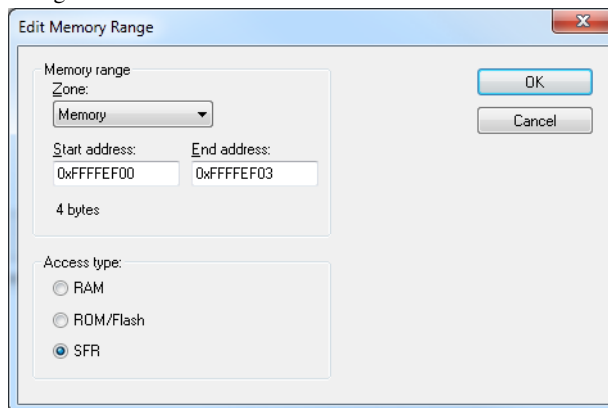
Deletes the selected memory area definition.

Delete All

Deletes all defined memory area definitions.

Edit Memory Range dialog box, for the C-SPY simulator

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



Use this dialog box to specify the memory areas, and their access types.

See also *Memory Configuration dialog box, for the C-SPY simulator*, page 156

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Memory range

Defines the memory area specific to your device:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 132.

Start address

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

End address

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

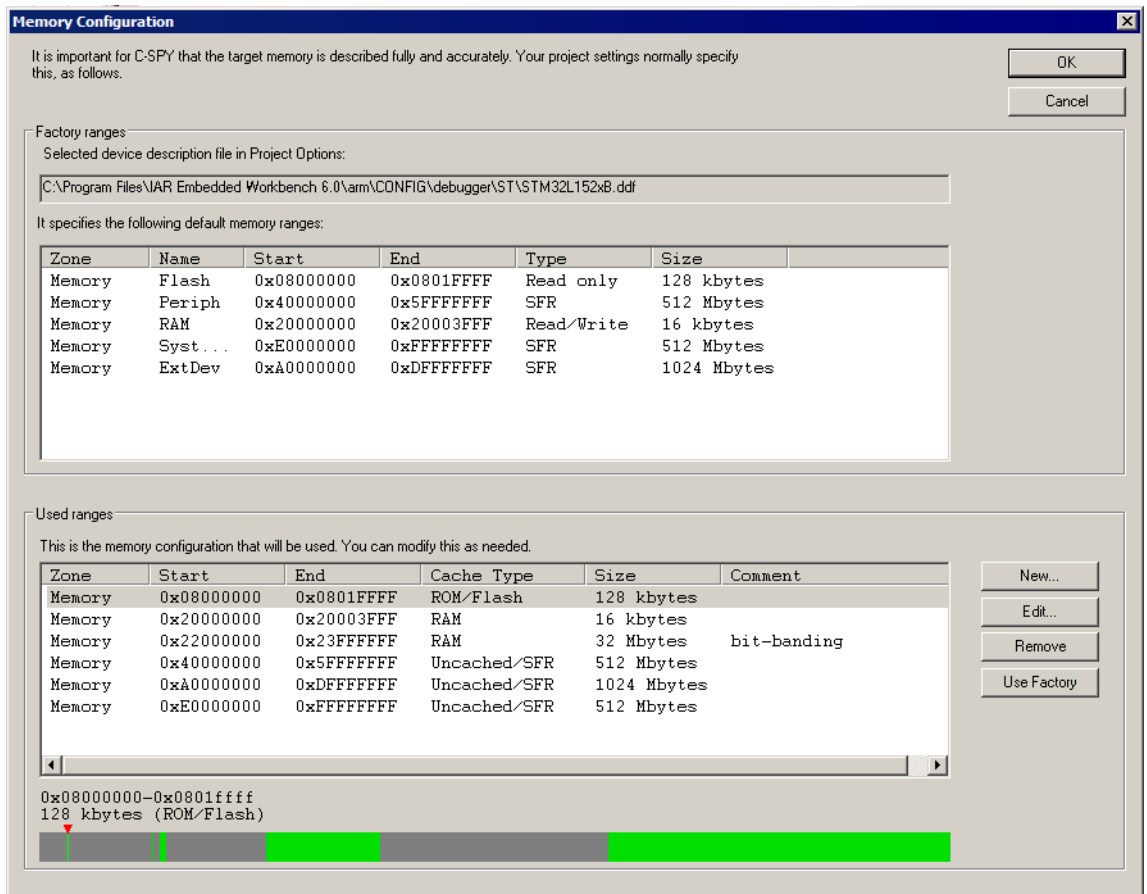
Access type

Selects an access type for the memory range; choose between:

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

Memory Configuration dialog box, in C-SPY hardware debugger drivers

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



C-SPY uses a default memory configuration based on information retrieved from the device description file that you select, or if memory configuration is missing in the device description file, tries to provide a usable factory default. See *Selecting a device description file*, page 39.

Use this dialog box to verify, and if needed, modify the memory areas so that they match the memory available on your device. Providing C-SPY with information about the memory layout of the target system is helpful both in terms of performance and functionality:

- Reading (and writing) memory (if your debug probe is connected through a USB port) can be fast, but is usually the limiting factor when C-SPY needs to update many debugger windows. Caching memory can speed up the performance, but then C-SPY needs information about the target memory.
- If C-SPY has been informed that the content of certain memory areas will be changed during a debug session, C-SPY can keep a copy of that memory readable even when the target does not normally allow reading (such as when executing).
- C-SPY can prevent accesses to areas without any memory at all, which can be important for certain hardware.

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

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

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Factory ranges

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

Zone

The memory zone, see *C-SPY memory zones*, page 132.

Name

The name of the memory area.

Start

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

End

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

Type

The access type of the memory area.

Size

The size of the memory area.

Used ranges

These columns lists the memory areas that you have specified manually:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 132.

Start

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

End

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

Cache Type

The cache type of the memory area.

Size

The size of the memory area.

Comment

Memory area information.

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

Graphical bar

A graphical bar that visualizes the whole theoretical memory space for the device. Defined areas are highlighted in green.

Buttons

These buttons are available:

New

Opens the **Edit Memory Range** dialog box, where you can specify a new memory area and attach a cache type to it, see *Edit Memory Range dialog box, for C-SPY hardware debugger drivers*, page 163.

Edit

Opens the **Edit Memory Range** dialog box, where you can edit the selected memory area. See *Edit Memory Range dialog box, for C-SPY hardware debugger drivers*, page 163.

Remove

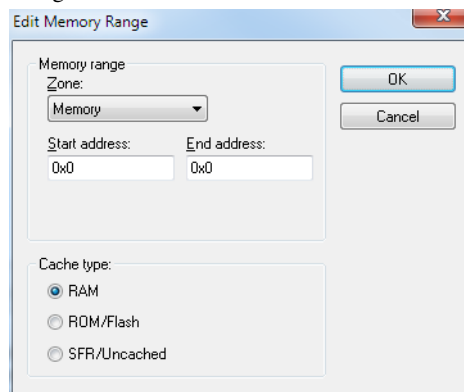
Removes the selected memory area definition.

Use Factory

Retrieves the memory areas as specified in the selected device description file, or if memory information is missing in the device description file, tries to provide a usable factory default.

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

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



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

See also *Memory Configuration dialog box, in C-SPY hardware debugger drivers*, page 160.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Memory range

Defines the memory area specific to your device:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 132.

Start address

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

End address

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

Cache type

Selects a cache type to the memory area; choose between:

RAM

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

ROM/Flash

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

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

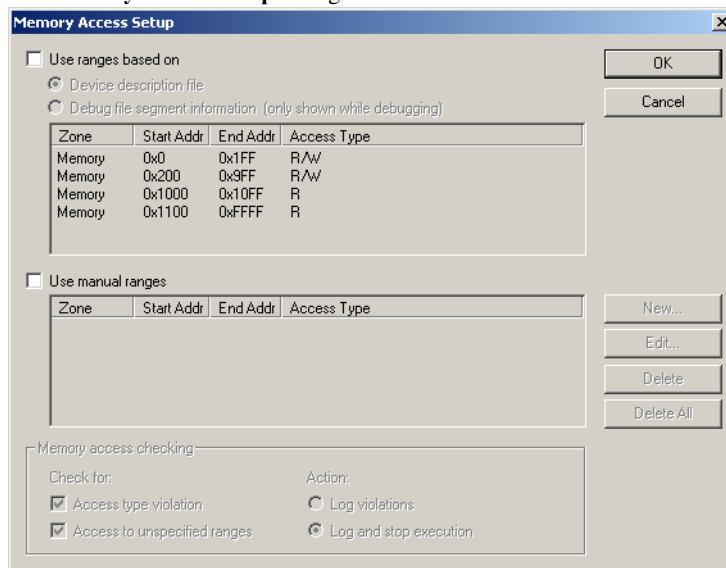
SFR/Uncached

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

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

Memory Access Setup dialog box

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



This dialog box lists all defined memory areas, where each column in the list specifies the properties of the area. In other words, the dialog box displays the memory access setup that will be used during the simulation.

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

For information about the columns and the properties displayed, see *Edit Memory Access dialog box*, page 167.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Use ranges based on

Selects any of the predefined alternatives for the memory access setup. Choose between:

Device description file

Loads properties from the device description file.

Debug file segment information

Properties are based on the segment information available in the debug file. This information is only available while debugging. The advantage of using this option, is that the simulator can catch memory accesses outside the linked application.

Use manual ranges

Specify your own ranges manually via the **Edit Memory Access** dialog box. To open this dialog box, choose **New** to specify a new memory range, or select a memory zone and choose **Edit** to modify it. For more information, see *Edit Memory Access dialog box*, page 167.

The ranges you define manually are saved between debug sessions.

Memory access checking

Check for determines what to check for;

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

Action selects the action to be performed if an access violation occurs; choose between:

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

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

Buttons

These buttons are available:

New

Opens the **Edit Memory Access** dialog box, where you can specify a new memory range and attach an access type to it, see *Edit Memory Access dialog box*, page 167.

Edit

Opens the **Edit Memory Access** dialog box, where you can edit the selected memory area. See *Edit Memory Access dialog box*, page 167.

Delete

Deletes the selected memory area definition.

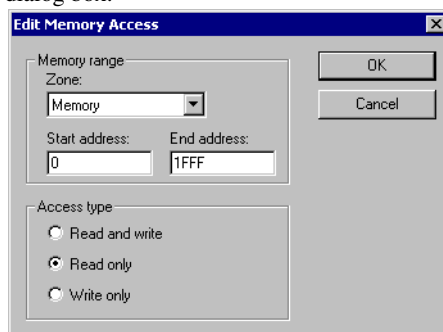
Delete All

Deletes all defined memory area definitions.

Note that except for the OK and Cancel buttons, buttons are only available when the option **Use manual ranges** is selected.

Edit Memory Access dialog box

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



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

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Memory range

Defines the memory area specific to your device:

Zone

Selects a memory zone, see *C-SPY memory zones*, page 132.

Start address

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

End address

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

Access type

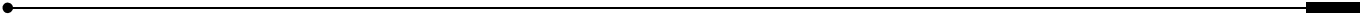
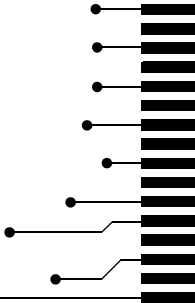
Selects an access type to the memory range; choose between:

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

Part 2. Analyzing your application

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

- Trace
- Profiling
- Code coverage





Trace

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

Introduction to using trace

These topics are covered:

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

See also:

- *Getting started using data logging*, page 77
- *Getting started using interrupt logging*, page 218
- *Profiling*, page 193

REASONS FOR USING TRACE

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

BRIEFLY ABOUT TRACE

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

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

Trace 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 **Interrupt Log**, **Interrupt Log Summary**, **Data Log**, and **Data Log Summary**.

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

REQUIREMENTS FOR USING TRACE

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

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

Collecting and using trace data

These tasks are covered:

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

GETTING STARTED WITH TRACE

- 1 Start C-SPY and choose *C-SPY driver*>**Trace Settings**. In the **Trace Settings** dialog box that is displayed, check if you need to change any of the default settings.

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



- 2 Open the **Trace** window—available from the driver-specific menu—and click the **Activate** button to enable collecting trace data.
- 3 Start the execution. When the execution stops, for example because a breakpoint is triggered, trace data is displayed in the **Trace** window. For more information about the window, see *Trace window*, page 174.

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 186 and *Trace Stop breakpoints dialog box*, page 187, 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 189.

- 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 190.

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 174
- *Function Trace window*, page 177
- *Timeline window*, page 177
- *Viewing Range dialog box*, page 185
- *Trace Start breakpoints dialog box*, page 186
- *Trace Stop breakpoints dialog box*, page 187
- *Trace Expressions window*, page 188
- *Find in Trace dialog box*, page 189
- *Find in Trace window*, page 190.

Trace window

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

This window displays the collected trace data.

The content of the **Trace** window depends on the C-SPY driver you are using.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Trace toolbar

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



Enable/Disable

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



Clear trace data

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



Toggle source

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



Browse

Toggles browse mode on or off for a selected item in the **Trace** window, see *Browsing through trace data*, page 174.



Find

Displays a dialog box where you can perform a search, see *Find in Trace dialog box*, page 189.



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 188.

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
5064	13582	00044F JC 0x043C	5
		DoForegroundProcess();	
5065	13588	00043C LCALL DoForegrou...	5
		DoForegroundProcess::?relay:	
5066	13594	000093 1B ; '.'	5
		?BDISPATCH_FF;	
5067	13597	000075 POP DPH	5
5068	13600	000077 POP DPL	5
5069	13604	000079 PUSH ?CBANK	5

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

#

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

Cycles

The number of cycles elapsed to this point.

Trace

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

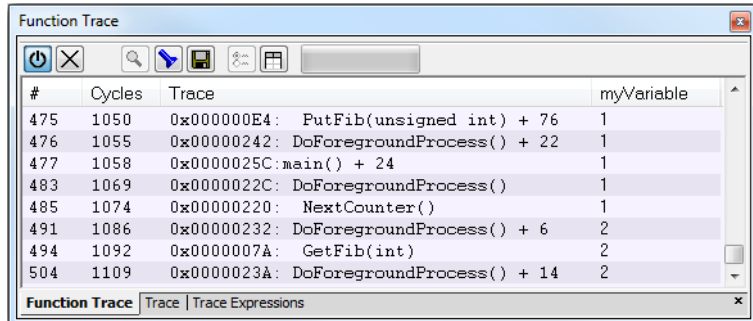
Expression

Each expression you have defined to be displayed appears in a separate column. Each entry in the expression column displays the value *after* executing the instruction on the same row. You specify the expressions for which you want to collect trace data in the **Trace Expressions** window, see *Trace Expressions window*, page 188.

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.

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 only shows trace data corresponding to calls to and returns from functions.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Toolbar

For information about the toolbar, see *Trace window*, page 174.

Display area

For information about the columns in the display area, see *Trace window*, page 174

Timeline window

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

This window displays trace data in different graphs in relation to a common time axis:

- Call Stack graph
- Data Log graph
- Interrupt Log graph

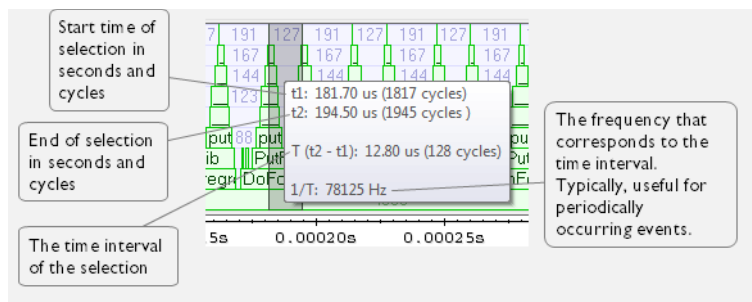
To display a graph:

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

- 2 In the **Timeline** window, click in the graph area and choose **Enable** from the context menu to enable a specific graph.
- 3 For the Data Log graph, you need to set a Data Log breakpoint for each variable you want a graphical representation of in the **Timeline** window. See *Data Log breakpoints dialog box*, page 124.
- 4 Click **Go** on the toolbar to start executing your application. The graph appears.

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

- Right-click and from the context menu choose **Zoom In** or **Zoom Out**.
Alternatively, use the + and - keys. The graph zooms in or out depending on which command you used.
- Right-click in the graph and from the context menu choose **Navigate** and the appropriate command to move backwards and forwards on the graph. Alternatively, use any of the shortcut keys: arrow keys, Home, End, and Ctrl+End.
- Double-click on a sample of interest and the corresponding source code is highlighted in the editor window and in the **Disassembly** window.
- Click on the graph and drag to select a time interval. Press Enter or right-click and from the context menu choose **Zoom>Zoom to Selection**. The selection zooms in. Point in the selection with the mouse pointer to get detailed tooltip information about the selected part of the graph:



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

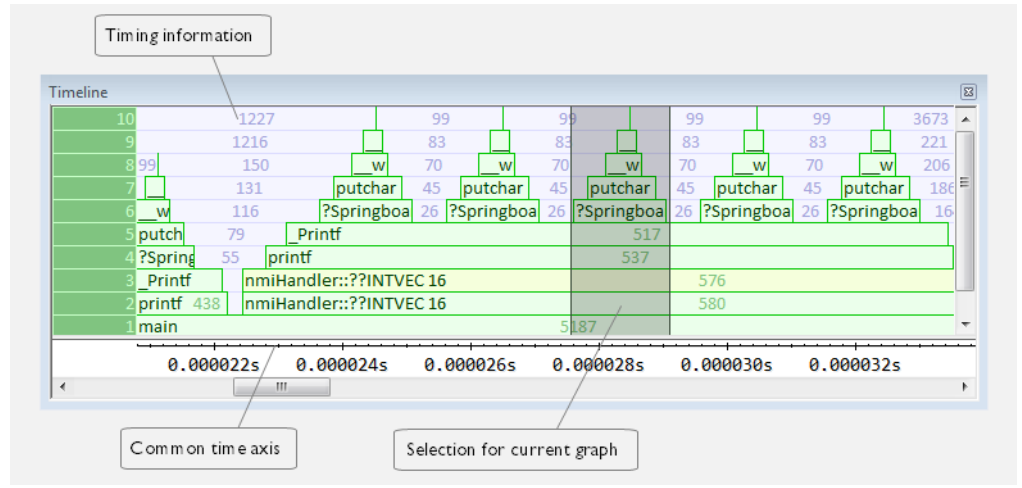
Requirements

See *Supported debugger features* in the release notes for information about applicability.

For more information about requirements related to trace data, see *Requirements for using trace*, page 172.

Display area for the Call Stack graph

The Call Stack graph displays the sequence of calls and returns collected by trace.



At the bottom of the graph you will usually find `main`, and above it, the functions called from `main`, and so on. The horizontal bars, which represent invocations of functions, use four different colors:

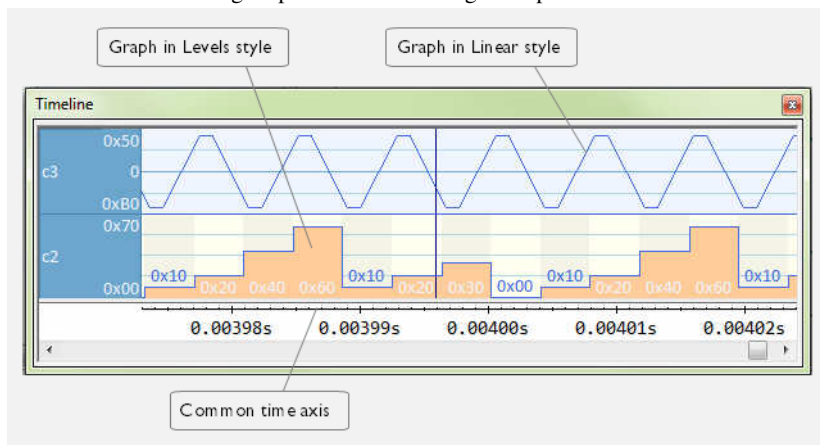
- Medium green for normal C functions with debug information
- Light green for functions known to the debugger only through an assembler label
- Medium or light yellow for interrupt handlers, with the same distinctions as for green.

The timing information represents the number of cycles spent in, or between, the function invocations.

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

Display area for the Data Log graph

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



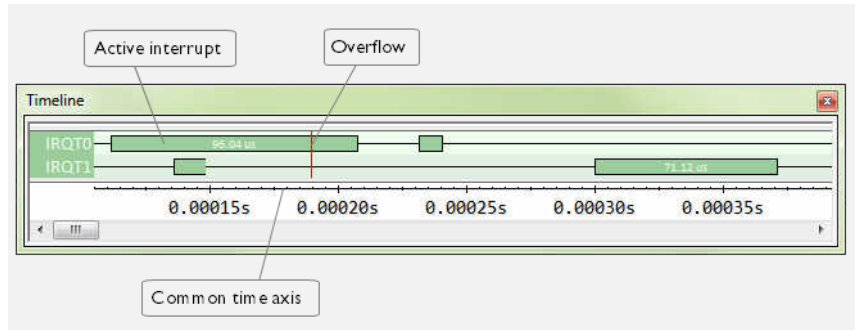
Where:

- The label area at the left end of the graph displays the variable name or the address for which you have specified the Data Log breakpoint.
- The graph itself displays how the value of the variable changes over time. The label area also displays the limits, or range, of the Y-axis for a variable. You can use the context menu to change these limits. The graph is a graphical representation of the information in the **Data Log** window, see *Data Log window*, page 95.
- The graph can be displayed either as a thin line between consecutive logs or as a rectangle for every log (optionally color-filled).
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all data logs from the target system. A red question mark indicates a log without a value.

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

Display area for the Interrupt Log graph

The Interrupt Log graph displays interrupts reported by the C-SPY simulator. In other words, the graph provides a graphical view of the interrupt events during the execution of your application.



Where:

- The label area at the left end of the graph displays the names of the interrupts.
- The graph itself shows active interrupts as a thick green horizontal bar where the white figure indicates the time spent in the interrupt. This graph is a graphical representation of the information in the **Interrupt Log** window, see *Interrupt Log window*, page 225.
- If the bar is displayed without horizontal borders, there are two possible causes:
 - The interrupt is reentrant and has interrupted itself. Only the innermost interrupt will have borders.
 - There are irregularities in the interrupt enter-leave sequence, probably due to missing logs.
- If the bar is displayed without a vertical border, the missing border indicates an approximate time for the log.
- A red vertical line indicates overflow, which means that the communication channel failed to transmit all interrupt logs from the target system.

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

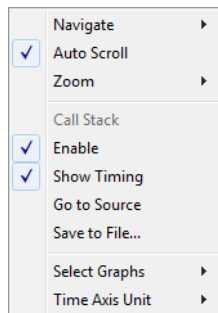
Selection and navigation

Click and drag to select. The selection extends vertically over all graphs, but appears highlighted in a darker color for the selected graph. You can navigate backward and forward in the selected graph using the left and right arrow keys. Use the Home and End

keys to move to the first or last relevant point, respectively. Use the navigation keys in combination with the Shift key to extend the selection.

Context menu

This context menu is available:



Note: The context menu contains some commands that are common to all graphs and some commands that are specific to each graph. The figure reflects the context menu for the Call Stack graph, which means that the menu looks slightly different for the other graphs.

These commands are available:

Navigate (All graphs)

Commands for navigating over the graph(s); choose between:

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

Previous moves the selection backward to the previous relevant point in the graph. Shortcut key: left arrow.

First moves the selection to the first data entry in the graph. Shortcut key: Home.

Last moves the selection to the last data entry in the graph. Shortcut key: End.

End moves the selection to the last data in any displayed graph, in other words the end of the time axis. Shortcut key: Ctrl+End.

Auto Scroll (All graphs)

Toggles auto scrolling on or off. When on, the most recently collected data is automatically displayed if you have executed the command **Navigate>End**.

Zoom (All graphs)

Commands for zooming the window, in other words, changing the time scale; choose between:

Zoom to Selection makes the current selection fit the window. Shortcut key: Return.

Zoom In zooms in on the time scale. Shortcut key: +.

Zoom Out zooms out on the time scale. Shortcut key: -.

10ns, 100ns, 1us, etc makes an interval of 10 nanoseconds, 100 nanoseconds, 1 microsecond, respectively, fit the window.

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

10m, 1h, etc makes an interval of 10 minutes or 1 hour, respectively, fit the window.

Data Log (Data Log graph)

A heading that shows that the Data Log-specific commands below are available.

Call Stack (Call Stack graph)

A heading that shows that the Call stack-specific commands below are available.

Interrupt (Interrupt Log graph)

A heading that shows that the Interrupt Log-specific commands below are available.

Enable (All graphs)

Toggles the display of the graph on or off. If you disable a graph, that graph will be indicated as **OFF** in the **Timeline** window. If no trace data has been collected for a graph, **no data** will appear instead of the graph.

Show Timing (Call Stack graph)

Toggles the display of the timing information on or off.

Variable (Data Log graph)

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

Solid Graph (Data Log graph)

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

Viewing Range (Data Log graph)

Displays a dialog box, see *Viewing Range dialog box*, page 185.

Size (Data Log graph)

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

Show Numerical Value (Data Log graph)

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

Go To Source (Common)

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

Save to File (Call Stack graph)

Saves all contents (or the selected contents) of the Call Stack graph to a file. The menu command is only available when C-SPY is not running.

Sort by (Interrupt graph)

Sorts the entries according to their ID or name. The selected order is used in the graph when new interrupts appear.

Select Graphs (Common)

Selects which graphs to be displayed in the **Timeline** window.

Time Axis Unit (Common)

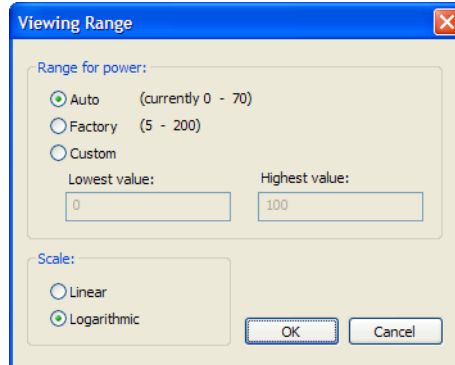
Selects the unit used in the time axis; choose between **Seconds** and **Cycles**.

Profile Selection

Enables profiling time intervals in the **Function Profiler** window. Note that this command is only available if the C-SPY driver supports PC Sampling.

Viewing Range dialog box

The **Viewing Range** dialog box is available from the context menu that appears when you right-click in the Data Log graph in the **Timeline** window.



Use this dialog box to specify the value range, that is, the range for the Y-axis for the graph.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Range for ...

Selects the viewing range for the displayed values:

Auto

Uses the range according to the range of the values that are actually collected, continuously keeping track of minimum or maximum values. The currently computed range, if any, is displayed in parentheses. The range is rounded to reasonably *even* limits.

Factory

For the Data Log graph: Uses the range according to the value range of the variable, for example 0–65535 for an unsigned 16-bit integer.

Custom

Use the text boxes to specify an explicit range.

Scale

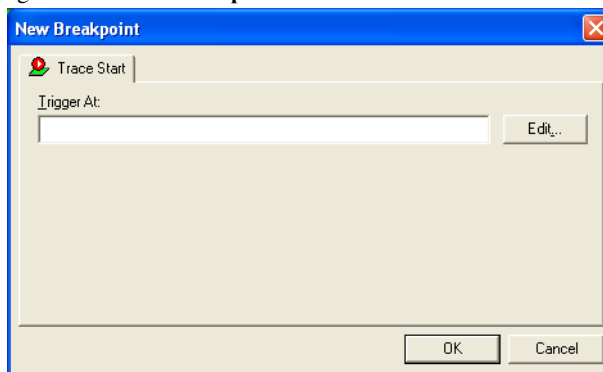
Selects the scale type of the Y-axis:

- **Linear**

- **Logarithmic.**

Trace Start breakpoints dialog box

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



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

See also, *Trace Stop breakpoints dialog box*, page 187.

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

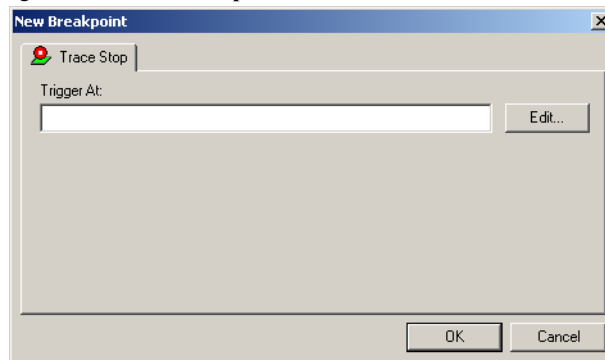
See *Supported debugger features* in the release notes for information about applicability.

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 127.

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 186.

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

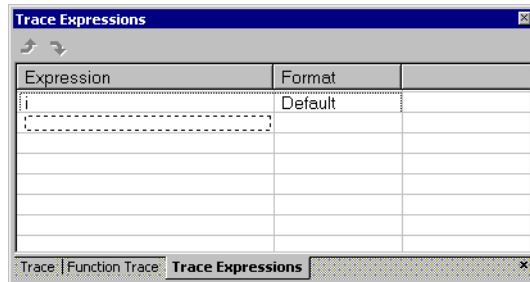
See *Supported debugger features* in the release notes for information about applicability.

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 127.

Trace Expressions window

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



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

Requirements

The C-SPY simulator.

Toolbar

The toolbar buttons change the order between the expressions:

Arrow up

Moves the selected row up.

Arrow down

Moves the selected row down.

Display area

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

Expression

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

Format

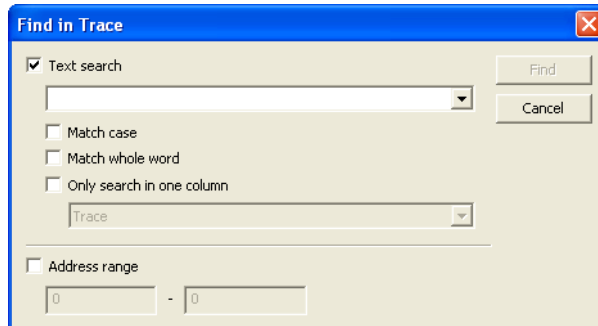
Shows which display format that is used for each expression. Note that you can change display format via the context menu.

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

Find in Trace dialog box

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

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



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

The search results are displayed in the **Find in Trace** window—available by choosing the **View>Messages** command, see *Find in Trace window*, page 190.

See also *Searching in trace data*, page 173.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

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

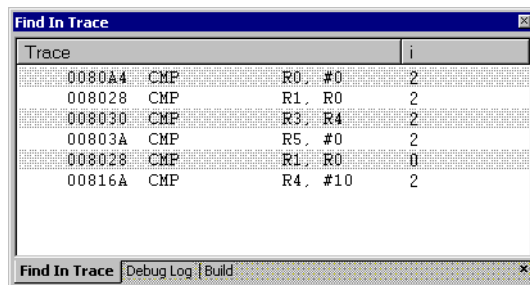
Searches only in the column you selected from the drop-down list.

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 189.

For more information, see *Searching in trace data*, page 173.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

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.

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*.

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

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

BRIEFLY ABOUT THE PROFILER

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

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

Profiling sources

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

- Trace (calls)
The full instruction trace is analyzed to determine all function calls and returns. When the collected instruction sequence is incomplete or discontinuous, the profiling information is less accurate.
- Trace (flat) / Sampling
Each instruction in the full instruction trace or each PC Sample is assigned to a corresponding function or code fragment, without regard to function calls or returns. This is most useful when the application does not exhibit normal call/return sequences, such as when you are using an RTOS, or when you are profiling code which does not have full debug information.
- Breakpoints
The profiler sets a breakpoint on every function entry point. During execution, the profiler collects information about function calls and returns as each breakpoint is hit. This assumes that the hardware supports a large number of breakpoints, and it has a huge impact on execution performance.

REQUIREMENTS FOR USING THE PROFILER

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

For information about the requirements for using the profiler, see *Supported debugger features* in the release notes.

Using the profiler

These tasks are covered:

- Getting started using the profiler on function level
- Analyzing the profiling data
- Getting started using the profiler on instruction level
- Selecting a time interval for profiling information

GETTING STARTED USING THE PROFILER ON FUNCTION LEVEL

To display function profiling information in the **Function Profiler** window:

- I Build your application using these options:

Category	Setting
C/C++ Compiler	Output>Generate debug information
Linker	Output>Format>Debug information for C-SPY (for the XLINK linker) Output>Include debug information in output (for the ILINK linker)

Table 5: 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, which is highlighted in blue. The menu options are: 'Enable' (checked), 'Clear', 'Source: Trace (calls)' (checked and circled in red), and 'Source: Trace (flat)'.

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.

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


SELECTING A TIME INTERVAL FOR PROFILING INFORMATION

Normally, the profiler computes its information from all PC samples it receives, accumulating more and more information until you explicitly clear the profiling information. However, you can choose a time interval for which the profiler computes the PC samples.

To select a time interval:

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

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

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

Reference information on the profiler

Reference information about:

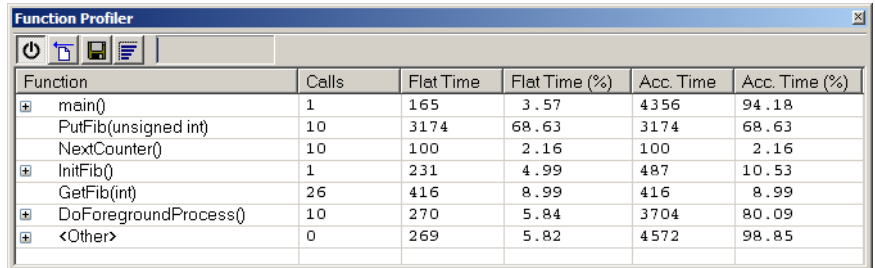
- *Function Profiler window*, page 199

See also:

- *Disassembly window*, page 60

Function Profiler window

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



Function	Calls	Flat Time	Flat Time (%)	Acc. Time	Acc. Time (%)
<input type="checkbox"/> main()	1	165	3.57	4356	94.18
<input type="checkbox"/> PutFib(unsigned int)	10	3174	68.63	3174	68.63
<input type="checkbox"/> NextCounter()	10	100	2.16	100	2.16
<input type="checkbox"/> InitFib()	1	231	4.99	487	10.53
<input type="checkbox"/> GetFib(int)	26	416	8.99	416	8.99
<input type="checkbox"/> DoForegroundProcess()	10	270	5.84	3704	80.09
<input type="checkbox"/> <Other>	0	269	5.82	4572	98.85

This window displays function profiling information.

When Trace(flat) is selected, a checkbox appears on each line in the left-side margin of the window. Use these checkboxes to include or exclude lines from the profiling. Excluded lines are dimmed but not removed.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Toolbar

The toolbar contains:



Enable/Disable

Enables or disables the profiler.

**Clear**

Clears all profiling data.

**Save**

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

**Graphical view**

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

Progress bar

Displays a backlog of profiling data that is still being processed. If the rate of incoming data is higher than the rate of the profiler processing the data, a backlog is accumulated. The progress bar indicates that the profiler is still processing data, but also approximately how far the profiler has come in the process. Note that because the profiler consumes data at a certain rate and the target system supplies data at another rate, the amount of data remaining to be processed can both increase and decrease. The progress bar can grow and shrink accordingly.

**Time-interval mode**

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

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 194.

Status field

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

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 194.

Display area

The content in the display area depends on which source that is used for the profiling information:

- For the Breakpoints and Trace (calls) sources, the display area contains one line for each function compiled with debug information enabled. When some profiling information has been collected, it is possible to expand rows of functions that have called other functions. The child items for a given function list all the functions that have been called by the parent function and the corresponding statistics.

- For the Sampling and Trace (flat) sources, the display area contains one line for each C function of your application, but also lines for sections of code from the runtime library or from other code without debug information, denoted only by the corresponding assembler labels. Each executed PC address from trace data is treated as a separate sample and is associated with the corresponding line in the Profiling window. Each line contains a count of those samples.

For information about which views that are supported in the C-SPY driver you are using, see *Requirements for using the profiler*, page 194.

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

Function (All sources)

The name of the profiled C function.

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

Calls (Breakpoints and Trace (calls))

The number of times the function has been called.

Flat time (Breakpoints and Trace (calls))

The time spent inside the function. The time depends on the C-SPY driver you are using and can be expressed as the number of executed instructions or expressed as the estimated number of cycles.

Flat time (%) (Breakpoints and Trace (calls))

Flat time expressed as a percentage of the total time.

Acc. time (Breakpoint and Trace (calls))

The time spent inside the function and everything called by the function. The time depends on the C-SPY driver you are using and can be expressed as the number of executed instructions or expressed as the estimated number of cycles.

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

Accumulated time expressed as a percentage of the total time.

PC Samples (Trace (flat) and Sampling)

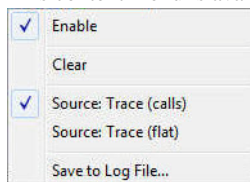
The number of PC samples associated with the function.

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

The number of PC samples associated with the function as a percentage of the total number of samples.

Context menu

This context menu is available:



The contents of this menu depend on the C-SPY driver you are using.

These commands are available:

Enable

Enables the profiler. The system will collect information also when the window is closed.

Clear

Clears all profiling data.

Filtering

Selects which part of your code to profile. Choose between:

Check All—Excludes all lines from the profiling.

Uncheck All—Includes all lines in the profiling.

Load—Reads all excluded lines from a saved file.

Save—Saves all excluded lines to a file. Typically, this can be useful if you are a group of engineers and want to share sets of exclusions.

These commands are only available when using one of the modes Trace(flat) or Sampling.

Source*

Selects which source to be used for the profiling information. Choose between:

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

Trace (calls)—the instruction count for instruction profiling is only as complete as the collected trace data.

Trace (flat)—the instruction count for instruction profiling is only as complete as the collected trace data.

Save to Log File

Saves all profiling data to a file.

* The available sources depend on the C-SPY driver you are using.

Code coverage

- Introduction to code coverage
- Reference information on code coverage.

Introduction to code coverage

These topics are covered:

- Reasons for using code coverage
- Briefly about code coverage
- Requirements and restrictions for using code coverage.

REASONS FOR USING CODE COVERAGE

The code coverage functionality is useful when you design your test procedure to verify whether all parts of the code have been executed. It also helps you identify parts of your code that are not reachable.

BRIEFLY ABOUT CODE COVERAGE

The **Code Coverage** window reports the status of the current code coverage analysis. For every program, module, and function, the analysis shows the percentage of code that has been executed since code coverage was turned on up to the point where the application has stopped. In addition, all statements that have not been executed are listed. The analysis will continue until turned off.

REQUIREMENTS AND RESTRICTIONS FOR USING CODE COVERAGE

Code coverage is supported by the C-SPY Simulator and there are no specific requirements or restrictions.

For information about support for code coverage in a hardware debugger system, see *Supported debugger features* in the release notes.

Reference information on code coverage

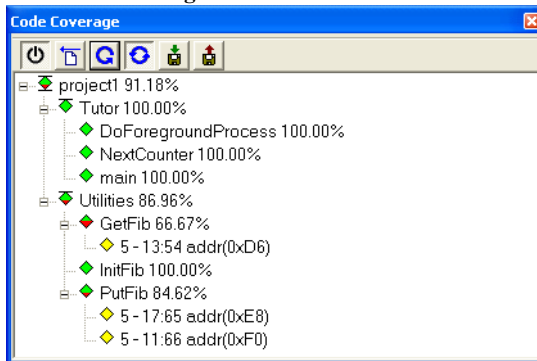
Reference information about:

- *Code Coverage window*, page 206.

See also *Single stepping*, page 54.

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 (for the XLINK linker)
Linker	Output>Include debug information in output (for the ILINK linker)
Debugger	Plugins>Code Coverage

Table 6: 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

See *Supported debugger features* in the release notes for information about applicability.

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:

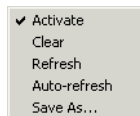
```
<column_start>-<column_end>: row address.
```

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.



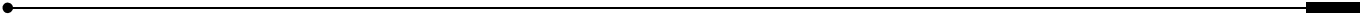
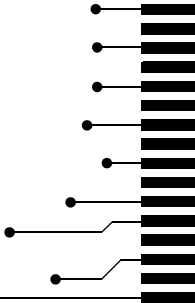
Restore session

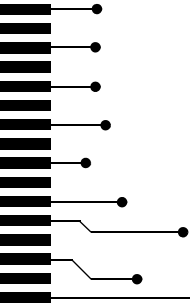
Restores previously saved code coverage session data. This is useful if you for some reason must abort your debug session, but want to continue the session later on. This command is available on the toolbar. This command might not be supported by the C-SPY driver you are using.

Part 3. Advanced debugging

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

- Interrupts
- C-SPY macros
- The C-SPY command line utility—`cspybat`
- The flash loading mechanism





Interrupts

- Introduction to interrupts
- Using the interrupt system
- Reference information on interrupts

Introduction to interrupts

These topics are covered:

- Briefly about interrupt logging
- Briefly about the interrupt simulation system
- Interrupt characteristics
- Interrupt simulation states
- C-SPY system macros for interrupt simulation
- Target-adapting the interrupt simulation system

See also:

- *Reference information on C-SPY system macros*, page 248
- *Breakpoints*, page 107
- *The IAR C/C++ Compiler Reference Guide*

BRIEFLY ABOUT INTERRUPT LOGGING

Interrupt logging provides you with comprehensive information about the interrupt events. This might be useful for example, to help you locate which interrupts you can fine-tune to become faster. You can log entrances and exits to and from interrupts. You can also log internal interrupt status information, such as triggered, expired, etc. In the IDE:

- The logs are displayed in the **Interrupt Log** window
- A summary is available in the **Interrupt Log Summary** window
- The Interrupt graph in the **Timeline** window provides a graphical view of the interrupt events during the execution of your application program.

Requirements for interrupt logging

Interrupt logging is supported by the C-SPY simulator.

See also *Getting started using interrupt logging*, page 218.

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 your microcontroller
- Single-occasion or periodical interrupts based on the cycle counter
- Predefined interrupts for various devices
- Configuration of hold time, probability, and timing variation
- State information for locating timing problems
- Configuration of interrupts using a dialog box or a C-SPY system macro—that is, one interactive and one automating interface. In addition, you can instantly force an interrupt.
- A log window that continuously displays events for each defined interrupt.
- A status window that shows the current interrupt activities.

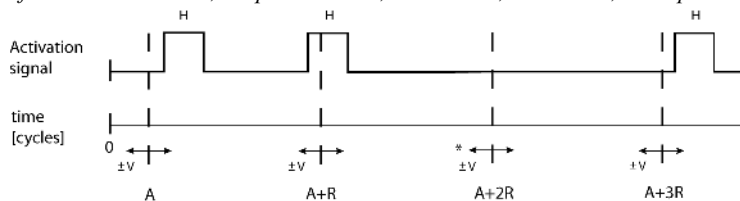
All interrupts you define using the **Interrupt Setup** dialog box are preserved between debug sessions, unless you remove them. A forced interrupt, on the other hand, exists only until it has been serviced and is not preserved between sessions.



The interrupt simulation system is activated by default, but if not required, you can turn off the interrupt simulation system to speed up the simulation. To turn it off, use either the **Interrupt Setup** dialog box or a system macro.

INTERRUPT CHARACTERISTICS

The simulated interrupts consist of a set of characteristics which lets you fine-tune each interrupt to make it resemble the real interrupt on your target hardware. You can specify a *first activation time*, a *repeat interval*, a *hold time*, a *variance*, and a *probability*.



* If probability is less than 100%, some interrupts may be omitted.

A = Activation time
 R = Repeat interval
 H = Hold time
 V = Variance

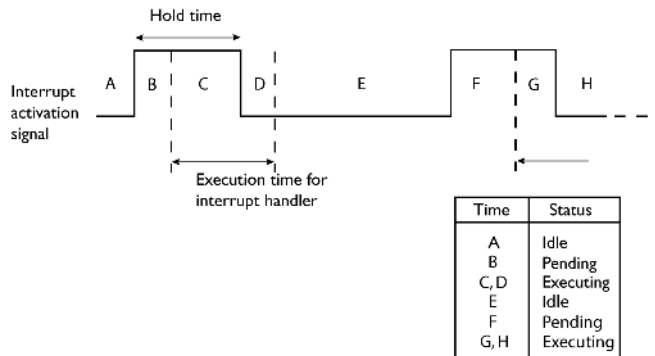
The interrupt simulation system uses the cycle counter as a clock to determine when an interrupt should be raised in the simulator. You specify the *first activation time*, which is based on the cycle counter. C-SPY will generate an interrupt when the cycle counter has passed the specified activation time. However, interrupts can only be raised between instructions, which means that a full assembler instruction must have been executed before the interrupt is generated, regardless of how many cycles an instruction takes.

To define the periodicity of the interrupt generation you can specify the *repeat interval* which defines the amount of cycles after which a new interrupt should be generated. In addition to the repeat interval, the periodicity depends on the two options *probability*—the probability, in percent, that the interrupt will actually appear in a period—and *variance*—a time variation range as a percentage of the repeat interval. These options make it possible to randomize the interrupt simulation. You can also specify a *hold time* which describes how long the interrupt remains pending until removed if it has not been processed. If the hold time is set to *infinite*, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

INTERRUPT SIMULATION STATES

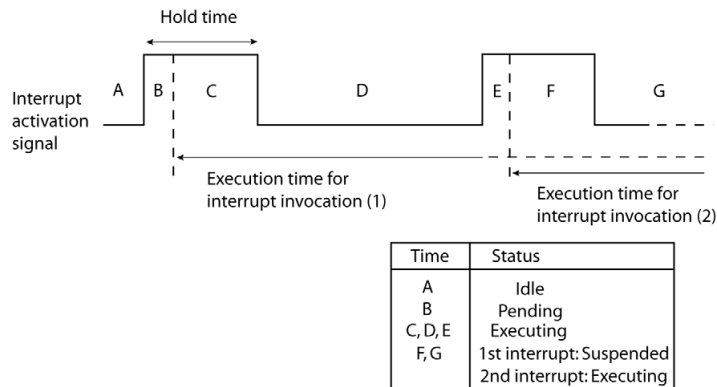
The interrupt simulation system contains status information that you can use for locating timing problems in your application. The **Interrupt Status** window displays the available status information. For an interrupt, these states can be displayed: *Idle*, *Pending*, *Executing*, or *Suspended*.

Normally, a repeatable interrupt has a specified repeat interval that is longer than the execution time. In this case, the status information at different times looks like this:



Note: The interrupt activation signal—also known as the pending bit—is automatically deactivated the moment the interrupt is acknowledged by the interrupt handler.

However, if the interrupt repeat interval is shorter than the execution time, and the interrupt is reentrant (or non-maskable), the status information at different times looks like this:



An execution time that is longer than the repeat interval might indicate that you should rewrite your interrupt handler and make it faster, or that you should specify a longer repeat interval for the interrupt simulation system.

C-SPY SYSTEM MACROS FOR INTERRUPT SIMULATION

Macros are useful when you already have sorted out the details of the simulated interrupt so that it fully meets your requirements. If you write a macro function containing definitions for the simulated interrupts, you can execute the functions automatically

when C-SPY starts. Another advantage is that your simulated interrupt definitions will be documented if you use macro files, and if you are several engineers involved in the development project you can share the macro files within the group.

The C-SPY Simulator provides these predefined system macros related to interrupts:

```
__enableInterrupts
__disableInterrupts
__orderInterrupt
__cancelInterrupt
__cancelAllInterrupts
__popSimulatorInterruptExecutingStack
```

The parameters of the first five macros correspond to the equivalent entries of the **Interrupts** dialog box.

For more information about each macro, see *Reference information on C-SPY system macros*, page 248.

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 resembles the behavior of the hardware, but only aspects of the interrupt that are relevant for the simulation are considered.

To simulate device-specific interrupts, the interrupt system must have detailed information about each available interrupt. This information is provided in the device description files.

For information about device description files, see *Selecting a device description file*, page 39.

Using the interrupt system

These tasks are covered:

- Simulating a simple interrupt
- Simulating an interrupt in a multi-task system
- Getting started using interrupt logging.

See also:

- *Using C-SPY macros*, page 235 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.

```
#pragma language = extended
#include <sdtio.h>
#include "iocpuname.h"
#include <intrinsic.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 = TIMER_VECTOR
__interrupt void basic_timer(void)
{
    ticks += 1;
}
```

- 2 Add your interrupt service routine to your application source code and add the file to your project.
- 3 Choose **Project>Options>Debugger>Setup** and select a device description file. The device description file contains information about the interrupt that C-SPY needs to be able to simulate it. Use the **Use device description file** browse button to locate the `ddf` file.
- 4 Build your project and start the simulator.

- 5 Choose **Simulator>Interrupt Setup** to open the **Interrupts Setup** dialog box. Select the **Enable interrupt simulation** option to enable interrupt simulation. Click **New** to open the **Edit Interrupt** dialog box. For the timer example, verify these settings:

Option	Settings
Interrupt	TIMER_VECTOR
First activation	4000
Repeat interval	2000
Hold time	10
Probability (%)	100
Variance (%)	0

Table 7: Timer interrupt settings

Click **OK**.

- 6 Execute your application. If you have enabled the interrupt properly in your application source code, C-SPY will:
- Generate an interrupt when the cycle counter has passed 4000
 - Continuously repeat the interrupt after approximately 2000 cycles.
- 7 To watch the interrupt in action, choose **Simulator>Interrupt Log** to open the **Interrupt Log** window.
- 8 From the context menu, available in the **Interrupt Log** window, choose **Enable** to enable the logging. If you restart program execution, status information about entrances and exits to and from interrupts will now appear in the **Interrupt Log** window.

For information about how to get a graphical representation of the interrupts correlated with a time axis, see *Timeline window*, page 177.

SIMULATING AN INTERRUPT IN A MULTI-TASK SYSTEM

If you are using interrupts in such a way that the normal instruction used for returning from an interrupt handler is not used, for example in an operating system with task-switching, the simulator cannot automatically detect that the interrupt has finished executing. The interrupt simulation system will work correctly, but the status information in the **Interrupt Setup** dialog box might not look as you expect. If too many interrupts are executing simultaneously, a warning might be issued.

To simulate a normal interrupt exit:

- I Set a code breakpoint on the instruction that returns from the interrupt function.

- 2 Specify the `__popSimulatorInterruptExecutingStack` macro as a condition to the breakpoint.

When the breakpoint is triggered, the macro is executed and then the application continues to execute automatically.

GETTING STARTED USING INTERRUPT LOGGING

- 1 Choose *C-SPY driver*>**Interrupt Log** to open the **Interrupt Log** window. Optionally, you can also choose:
 - *C-SPY driver*>**Interrupt Log Summary** to open the **Interrupt Log Summary** window
 - *C-SPY driver*>**Timeline** to open the **Timeline** window and view the Interrupt graph.
- 2 From the context menu in the **Interrupt Log** window, choose **Enable** to enable the logging.
- 3 Start executing your application program to collect the log information.
- 4 To view the interrupt log information, look in any of the Interrupt Log, Interrupt Log Summary, or the Interrupt graph in the **Timeline** window.
- 5 If you want to save the log or summary to a file, choose **Save to log file** from the context menu in the window in question.
- 6 To disable interrupt logging, from the context menu in the **Interrupt Log** window, toggle **Enable** off.

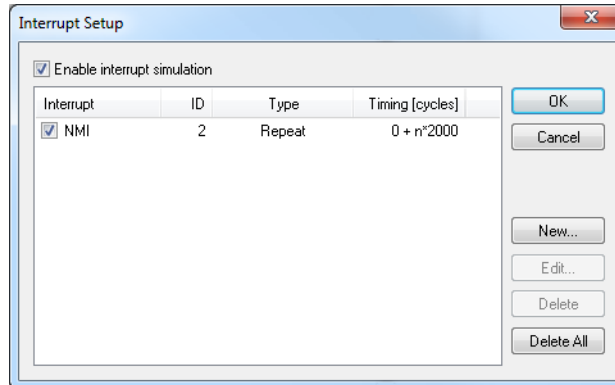
Reference information on interrupts

Reference information about:

- *Interrupt Setup dialog box*, page 219
- *Edit Interrupt dialog box*, page 220
- *Forced Interrupt window*, page 222
- *Interrupt Status window*, page 223
- *Interrupt Log window*, page 225
- *Interrupt Log Summary window*, page 228.

Interrupt Setup dialog box

The **Interrupt Setup** dialog box is available by choosing **Simulator>Interrupt Setup**.



This dialog box lists all defined interrupts. Use this dialog box to enable or disable the interrupt simulation system, as well as to enable or disable individual interrupts.

Requirements

The C-SPY simulator.

Enable interrupt simulation

Enables or disables interrupt simulation. If the interrupt simulation is disabled, the definitions remain but no interrupts are generated. Note that you can also enable and disable installed interrupts individually by using the check box to the left of the interrupt name in the list of installed interrupts.

Display area

This area contains these columns:

Interrupt

Lists all interrupts. Use the checkbox to enable or disable the interrupt.

ID

A unique interrupt identifier.

Type

Shows the type of the interrupt. The type can be one of:

Forced, a single-occasion interrupt defined in the Forced Interrupt Window.

Single, a single-occasion interrupt.

Repeat, a periodically occurring interrupt.

If the interrupt has been set from a C-SPY macro, the additional part (**macro**) is added, for example: **Repeat(macro)**.

Timing

The timing of the interrupt. For a **Single** and **Forced** interrupt, the activation time is displayed. For a **Repeat** interrupt, the information has the form: `Activation Time + n*Repeat Time`. For example, `2000 + n*2345`. This means that the first time this interrupt is triggered, is at 2000 cycles and after that with an interval of 2345 cycles.

Buttons

These buttons are available:

New

Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 220.

Edit

Opens the **Edit Interrupt** dialog box, see *Edit Interrupt dialog box*, page 220.

Delete

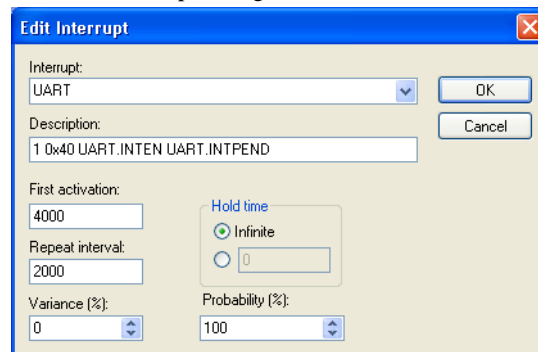
Removes the selected interrupt.

Delete All

Removes all interrupts.

Edit Interrupt dialog box

The **Edit Interrupt** dialog box is available from the **Interrupt Setup** dialog box.



Use this dialog box to interactively fine-tune the interrupt parameters. You can add the parameters and quickly test that the interrupt is generated according to your needs.

Note: You can only edit or remove non-forced interrupts.

Requirements

The C-SPY simulator.

Interrupt

Selects the interrupt that you want to edit. The drop-down list contains all available interrupts. Your selection will automatically update the **Description** box. The list is populated with entries from the device description file that you have selected.

Description

A description of the selected interrupt, if available. The description is retrieved from the selected device description file. For interrupts specified using the system macro `__orderInterrupt`, the **Description** box is empty.

First activation

Specify the value of the cycle counter after which the specified type of interrupt will be generated.

Repeat interval

Specify the periodicity of the interrupt in cycles.

Variance %

Selects a timing variation range, as a percentage of the repeat interval, in which the interrupt might occur for a period. For example, if the repeat interval is 100 and the variance 5%, the interrupt might occur anywhere between $T=95$ and $T=105$, to simulate a variation in the timing.

Hold time

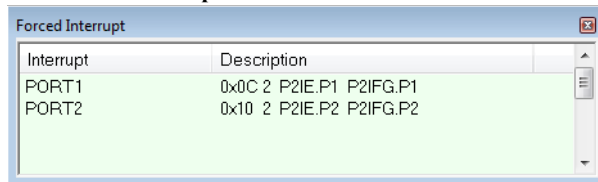
Specify how long, in cycles, the interrupt remains pending until removed if it has not been processed. If you select **Infinite**, the corresponding pending bit will be set until the interrupt is acknowledged or removed.

Probability %

Selects the probability, in percent, that the interrupt will actually occur within the specified period.

Forced Interrupt window

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



Use this window to force an interrupt instantly. This is useful when you want to check your interrupt logic and interrupt routines. Just start typing an interrupt name and focus shifts to the first line found with that name.

The hold time for a forced interrupt is infinite, and the interrupt exists until it has been serviced or until a reset of the debug session.

To sort the window contents, click on either the **Interrupt** or the **Description** column header. A second click on the same column header reverses the sort order.

To force an interrupt:

- 1 Enable the interrupt simulation system, see *Interrupt Setup dialog box*, page 219.
- 2 Double-click the interrupt in the **Forced Interrupt** window, or activate by using the **Force** command available on the context menu.

Requirements

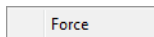
The C-SPY simulator.

Display area

This area lists all available interrupts and their definitions. This information is retrieved from the selected device description file. See this file for a detailed description.

Context menu

This context menu is available:



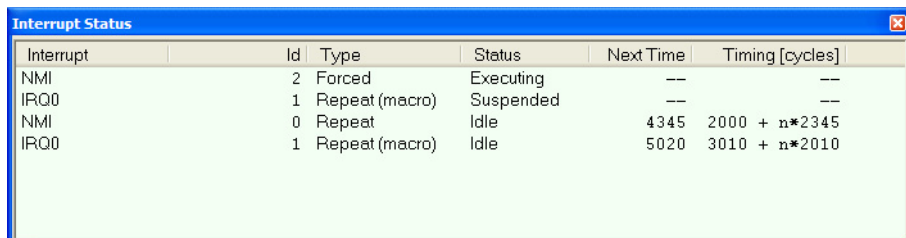
This command is available:

Force

Triggers the interrupt you selected in the display area.

Interrupt Status window

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



Interrupt	Id	Type	Status	Next Time	Timing [cycles]
NMI	2	Forced	Executing	--	--
IRQ0	1	Repeat (macro)	Suspended	--	--
NMI	0	Repeat	Idle	4345	2000 + n*2345
IRQ0	1	Repeat (macro)	Idle	5020	3010 + n*2010

This window shows the status of all the currently active interrupts, in other words interrupts that are either executing or waiting to be executed.

Requirements

The C-SPY simulator.

Display area

This area contains these columns:

Interrupt

Lists all interrupts.

ID

A unique interrupt identifier.

Type

The type of the interrupt. The type can be one of:

Forced, a single-occasion interrupt defined in the Forced Interrupt window.

Single, a single-occasion interrupt.

Repeat, a periodically occurring interrupt.

If the interrupt has been set from a C-SPY macro, the additional part (**macro**) is added, for example: **Repeat(macro)**.

Status

The state of the interrupt:

Idle, the interrupt activation signal is low (deactivated).

Pending, the interrupt activation signal is active, but the interrupt has not been yet acknowledged by the interrupt handler.

Executing, the interrupt is currently being serviced, that is the interrupt handler function is executing.

Suspended, the interrupt is currently suspended due to execution of an interrupt with a higher priority.

(deleted) is added to **Executing** and **Suspended** if you have deleted a currently active interrupt. **(deleted)** is removed when the interrupt has finished executing.

Next Time

The next time an idle interrupt is triggered. Once a repeatable interrupt starts executing, a copy of the interrupt will appear with the state Idle and the next time set. For interrupts that do not have a next time—that is pending, executing, or suspended—the column will show --.

Timing

The timing of the interrupt. For a **Single** and **Forced** interrupt, the activation time is displayed. For a **Repeat** interrupt, the information has the form: $\text{Activation Time} + n * \text{Repeat Time}$. For example, $2000 + n * 2345$. This means that the first time this interrupt is triggered, is at 2000 cycles and after that with an interval of 2345 cycles.

Interrupt Log window

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

Time	Interrupt	Status	Program Counter	Execution Time
109.32 us	IRQ0	Triggered	0x13E8	
111.26 us	IRQ0	Enter	0x13F0	
135.78 us	IRQ1	Enter	0x1126	
148.72 us	IRQ1	Leave	0x1378	12.94 us
189.34 us	Overflow			
207.30 us	IRQ0	Leave	0x1126	96.04 us
230.00 us	IRQ0	Triggered	0x1110	
231.34 us	IRQ0	Enter	0x1126	
240.26 us	IRQ0	Leave	0x1122	8.92 us
300.00 us	IRQ1	Enter	---	
371.12 us	IRQ1	Leave	0x1120	71.12 us

Red indicates overflows and italic indicates approximate values

Light-colored rows indicate entrances to interrupts

Darker rows indicate exits from interrupts

This window logs entrances to and exits from interrupts. The C-SPY simulator also logs internal state changes.

The information is useful for debugging the interrupt handling in the target system. When the **Interrupt Log** window is open, it is updated continuously at runtime.

Note: There is a limit on the number of saved logs. When this limit is exceeded, the entries in the beginning of the buffer are erased.

For more information, see *Getting started using interrupt logging*, page 218.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 177.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Display area

This area contains these columns:

Time

The time for the interrupt entrance, based on an internally specified clock frequency.

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

Cycles

The number of cycles from the start of the execution until the event.

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

Interrupt

The interrupt as defined in the device description file.

Status

Shows the event status of the interrupt:

Triggered, the interrupt has passed its activation time.

Forced, the same as Triggered, but the interrupt was forced from the Forced Interrupt window.

Enter, the interrupt is currently executing.

Leave, the interrupt has been executed.

Expired, the interrupt hold time has expired without the interrupt being executed.

Rejected, the interrupt has been rejected because the necessary interrupt registers were not set up to accept the interrupt.

Program Counter

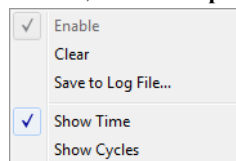
The value of the program counter when the event occurred.

Execution Time/Cycles

The time spent in the interrupt, calculated using the Enter and Leave timestamps. This includes time spent in any subroutines or other interrupts that occurred in the specific interrupt.

Context menu

This context menu is available in the **Data Log** window, the **Data Log Summary** window, the **Interrupt Log** window, and in the **Interrupt Log Summary** window:



Note: The commands are the same in each window, but they only operate on the specific window.

These commands are available:

Enable

Enables the logging system. The system will log information also when the window is closed.

Clear

Deletes the log information. Note that this will happen also when you reset the debugger.

Save to log file

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

Show Time

Displays the **Time** column in the **Data Log** window and in the **Interrupt Log** window, respectively.

This menu command might not be available in the C-SPY driver you are using, which means that the **Time** column is by default displayed in the **Data Log** window.

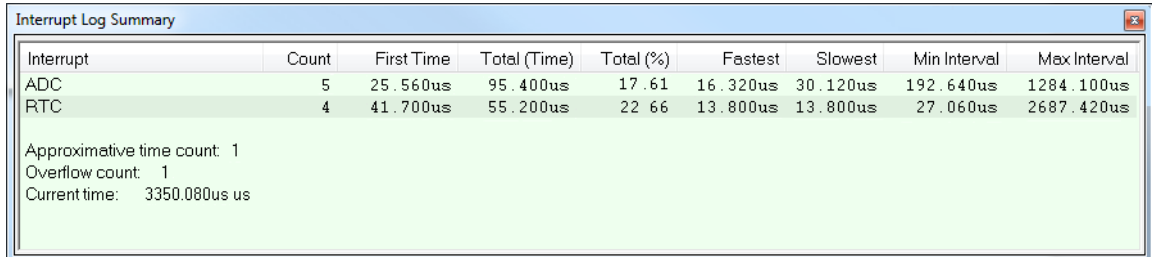
Show Cycles

Displays the **Cycles** column in the **Data Log** window and in the **Interrupt Log** window, respectively.

This menu command might not be available in the C-SPY driver you are using, which means that the **Cycles** column is not supported.

Interrupt Log Summary window

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



Interrupt	Count	First Time	Total (Time)	Total (%)	Fastest	Slowest	Min Interval	Max Interval
ADC	5	25.560us	95.400us	17.61	16.320us	30.120us	192.640us	1284.100us
RTC	4	41.700us	55.200us	22.66	13.800us	13.800us	27.060us	2687.420us

Approximative time count: 1
 Overflow count: 1
 Current time: 3350.080us us

This window displays a summary of logs of entrances to and exits from interrupts.

For more information, see *Getting started using interrupt logging*, page 218.

For information about how to get a graphical view of the interrupt events during the execution of your application, see *Timeline window*, page 177.

Requirements

See *Supported debugger features* in the release notes for information about applicability.

Display area

Each row in this area displays statistics about the specific interrupt based on the log information in these columns; and summary information is listed at the bottom of the display area:

Interrupt

The type of interrupt that occurred.

Count

The number of times the interrupt occurred.

First time

The first time the interrupt was executed.

Total (Time)**

The accumulated time spent in the interrupt.

Total (%)

The time in percent of the current time.

Fastest**

The fastest execution of a single interrupt of this type.

Slowest**

The slowest execution of a single interrupt of this type.

Min interval

The shortest time between two interrupts of this type.

The interval is specified as the time interval between the entry time for two consecutive interrupts.

Max interval

The longest time between two interrupts of this type.

The interval is specified as the time interval between the entry time for two consecutive interrupts.

Approximative time count

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

For some C-SPY drivers, this information is not displayed or the value is always zero. In this case, all logs have an exact time stamp.

For other C-SPY drivers, a non-zero value is displayed. The value represents the amount of logs with an approximative time stamp. This might happen if the bandwidth in the communication channel is too low compared to the amount of data packets generated by the CPU or if the CPU generated packets with an approximative time stamp.

Overflow count

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

For some C-SPY drivers, this information is not displayed or the value is always zero.

For other C-SPY drivers, the number represents the amount of overflows in the communication channel which can cause logs to be lost. If this happens, it indicates that logs might be incomplete. To solve this, make sure not to use all C-SPY log features simultaneously or check used bandwidth for the communication channel.

Current time|cycles

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

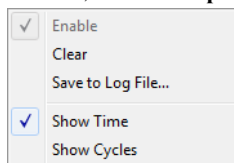
For some C-SPY drivers, the value is always zero or not visible at all.

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

** Calculated in the same way as for the Execution time/cycles in the **Interrupt Log** window.

Context menu

This context menu is available in the **Data Log** window, the **Data Log Summary** window, the **Interrupt Log** window, and in the **Interrupt Log Summary** window:



Note: The commands are the same in each window, but they only operate on the specific window.

These commands are available:

Enable

Enables the logging system. The system will log information also when the window is closed.

Clear

Deletes the log information. Note that this will happen also when you reset the debugger.

Save to log file

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

Show Time

Displays the **Time** column in the **Data Log** window and in the **Interrupt Log** window, respectively.

This menu command might not be available in the C-SPY driver you are using, which means that the **Time** column is by default displayed in the **Data Log** window.

Show Cycles

Displays the **Cycles** column in the **Data Log** window and in the **Interrupt Log** window, respectively.

This menu command might not be available in the C-SPY driver you are using, which means that the **Cycles** column is not supported.

C-SPY macros

- Introduction to C-SPY macros
- Using C-SPY macros
- Reference information on the macro language
- Reference information on reserved setup macro function names
- Reference information on C-SPY system macros
- Graphical environment for macros

Introduction to C-SPY macros

These topics are covered:

- Reasons for using C-SPY macros
- Briefly about using C-SPY macros
- Briefly about setup macro functions and files
- Briefly about the macro language

REASONS FOR USING C-SPY MACROS

You can use C-SPY macros either by themselves or in conjunction with complex breakpoints and interrupt simulation to perform a wide variety of tasks. Some examples where macros can be useful:

- Automating the debug session, for instance with trace printouts, printing values of variables, and setting breakpoints.
- Hardware configuring, such as initializing hardware registers.
- Feeding your application with simulated data during runtime.
- Simulating peripheral devices, see the chapter *Interrupts*. This only applies if you are using the simulator driver.
- Developing small debug utility functions, for instance reading I/O input from a file, see the file `setupsimple.mac` located in the directory `\cpuname\tutor\`.

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 245.

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 240.

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 43.

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 237.
- You can register macros interactively in the **Macro Registration** window, see *Macro Registration window*, page 281. Registered macros appear in the **Debugger Macros** window, see *Debugger Macros window*, page 283.
- 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 266.

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 237.
- 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 237.
- 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 285.
- 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 238.

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

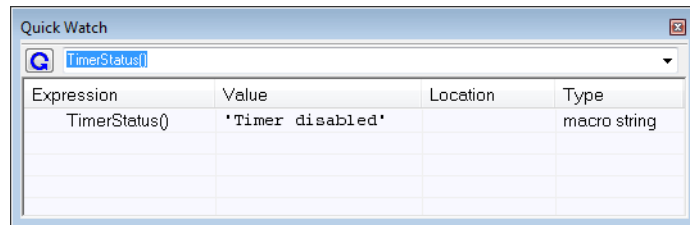
The **Quick Watch** window lets you dynamically choose when to execute a macro function.

- 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 90.

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 115.
- 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 243.

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 240
- *Macro variables*, page 240
- *Macro parameters*, page 241
- *Macro strings*, page 241
- *Macro statements*, page 242
- *Formatted output*, page 243.

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 72.

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 8: 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 `cs Spybat`).
- 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 294.

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 243.

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 72.

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 261.

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 234.

Reference information about:

- `execUserPreload`
- `execUserExecutionStarted`
- `execUserExecutionStopped`
- `execUserFlashInit`
- `execUserSetup`
- `execUserFlashReset`
- `execUserPreReset`
- `execUserReset`
- `execUserExit`
- `execUserFlashExit`

execUserPreload

Syntax	<code>execUserPreload</code>
For use with	All C-SPY drivers.
Description	Called after communication with the target system is established but before downloading the target application.

Implement this macro to initialize memory locations and/or registers which are vital for loading data properly.

execUserExecutionStarted

Syntax	<code>execUserExecutionStarted</code>
For use with	See <i>Supported debugger features</i> in the release notes for information about applicability.
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	See <i>Supported debugger features</i> in the release notes for information about applicability.
Description	Called when the debugger has stopped execution. The macro is not called when performing a one-instruction assembler step, in other words, Step or Step Into in the Disassembly window.

execUserFlashInit

Syntax	<code>execUserFlashInit</code>
For use with	See <i>Supported debugger features</i> in the release notes for information about applicability.
Description	Called once before the flash loader is downloaded to RAM. Implement this macro typically for setting up the memory map required by the flash loader. This macro is only called when you are programming flash, and it should only be used for flash loader functionality.

execUserSetup

Syntax	<code>execUserSetup</code>
--------	----------------------------

For use with	All C-SPY drivers.
Description	<p>Called once after the target application is downloaded.</p> <p>Implement this macro to set up the memory map, breakpoints, interrupts, register macro files, etc.</p> <p> If you define interrupts or breakpoints in a macro file that is executed at system start (using <code>execUserSetup</code>) we strongly recommend that you also make sure that they are removed at system shutdown (using <code>execUserExit</code>). An example is available in <code>SetupSimple.mac</code>, see the tutorials in the Information Center.</p> <p>The reason for this is that the simulator saves interrupt settings between sessions and if they are not removed they will get duplicated every time <code>execUserSetup</code> is executed again. This seriously affects the execution speed.</p>

execUserFlashReset

Syntax	<code>execUserFlashReset</code>
For use with	See <i>Supported debugger features</i> in the release notes for information about applicability.
Description	Called once after the flash loader is downloaded to RAM, but before execution of the flash loader. This macro is only called when you are programming flash, and it should only be used for flash loader functionality.

execUserPreReset

Syntax	<code>execUserPreReset</code>
For use with	All C-SPY drivers.
Description	<p>Called each time just before the reset command is issued.</p> <p>Implement this macro to set up any required device state.</p>

execUserReset

Syntax	<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 `execUserExit`

For use with All C-SPY drivers.

Description Called once when the debug session ends.
Implement this macro to save status data etc.

execUserFlashExit

Syntax `execUserFlashExit`

For use with See *Supported debugger features* in the release notes for information about applicability.

Description Called once when the flash programming ends.
Implement this macro to save status data etc. This macro is useful for flash loader functionality.

Reference information on C-SPY system macros

This section gives reference information about each of the C-SPY system macros.

This table summarizes the pre-defined system macros:

Macro	Description
<code>__cancelAllInterrupts</code>	Cancels all ordered interrupts
<code>__cancelInterrupt</code>	Cancels an interrupt
<code>__clearBreak</code>	Clears a breakpoint
<code>__closeFile</code>	Closes a file that was opened by <code>__openFile</code>
<code>__delay</code>	Delays execution
<code>__disableInterrupts</code>	Disables generation of interrupts
<code>__driverType</code>	Verifies the driver type

Table 9: Summary of system macros

Macro	Description
<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>__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>__memorySave</code>	Saves the contents of a specified memory area to a file
<code>__messageBoxYesCancel</code>	Displays a Yes/Cancel dialog box for user interaction
<code>__messageBoxYesNo</code>	Displays a Yes/No dialog box for user interaction
<code>__openFile</code>	Opens a file for I/O operations
<code>__orderInterrupt</code>	Generates an interrupt
<code>__popSimulatorInterruptExecutingStack</code>	Informs the interrupt simulation system that an interrupt handler has finished executing
<code>__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>__setDataBreak</code>	Sets a data breakpoint
<code>__setDataLogBreak</code>	Sets a data log breakpoint
<code>__setLogBreak</code>	Sets a log breakpoint
<code>__setSimBreak</code>	Sets a simulation breakpoint

Table 9: Summary of system macros (Continued)

Macro	Description
<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>__toLowerCase</code>	Returns a copy of the parameter string where all the characters have been converted to lower case
<code>__toString</code>	Prints strings
<code>__toUpperCase</code>	Returns a copy of the parameter string where all the characters have been converted to upper case
<code>__unloadImage</code>	Unloads a debug image
<code>__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 9: Summary of system macros (Continued)

__cancelAllInterrupts

Syntax	<code>__cancelAllInterrupts()</code>
Return value	<code>int 0</code>
For use with	The C-SPY Simulator.
Description	Cancels all ordered interrupts.

__cancelInterrupt

Syntax `__cancelInterrupt(interrupt_id)`

Parameters *interrupt_id*

The value returned by the corresponding `__orderInterrupt` macro call (unsigned long).

Return value

Result	Value
Successful	int 0
Unsuccessful	Non-zero error number

Table 10: `__cancelInterrupt` return values

For use with The C-SPY Simulator.

Description Cancels the specified interrupt.

__clearBreak

Syntax `__clearBreak(break_id)`

Parameters *break_id*

The value returned by any of the set breakpoint macros.

Return value int 0

For use with All C-SPY drivers.

Description Clears a user-defined breakpoint.

See also *Breakpoints*, page 107.

__closeFile

Syntax `__closeFile(fileHandle)`

Parameters *fileHandle*

A macro variable used as filehandle by the `__openFile` macro.

Return value int 0

For use with All C-SPY drivers.

Description Closes a file previously opened by `__openFile`.

`__delay`

Syntax `__delay(value)`

Parameters *value*
The number of milliseconds to delay execution.

Return value `int 0`

For use with All C-SPY drivers.

Description Delays execution the specified number of milliseconds.

`__disableInterrupts`

Syntax `__disableInterrupts()`

Return value

Result	Value
Successful	<code>int 0</code>
Unsuccessful	Non-zero error number

Table 11: `__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.

"*string*" corresponds to the C-SPY driver. See *Supported debugger features* in the release notes for information about possible strings.

Return value

Result	Value
Successful	1
Unsuccessful	0

Table 12: `__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 13: `__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 14: `__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 132.

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 <i>value</i> and the existing contents of memory before writing the result to memory.
Return value	int 0	
For use with	All C-SPY drivers.	
Description	Fills a specified memory area with a byte value.	
Example	<code>__fillMemory8(0x80, 0x700, "", 0x10, "OR");</code>	

__fillMemory16

Syntax	<code>__fillMemory16(<i>value</i>, <i>address</i>, <i>zone</i>, <i>length</i>, <i>format</i>)</code>	
Parameters	<i>value</i>	An integer that specifies the value.
	<i>address</i>	An integer that specifies the memory start address.
	<i>zone</i>	A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 132.
	<i>length</i>	An integer that defines how many 2-byte entities to be affected.
	<i>format</i>	One of these alternatives:
	Copy	<i>value</i> will be copied to the specified memory area.
	AND	An AND operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
	OR	An OR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
	XOR	An XOR operation will be performed between <i>value</i> and the existing contents of memory before writing the result to memory.
Return value	int 0	

For use with	All C-SPY drivers.
Description	Fills a specified memory area with a 2-byte value.
Example	<code>__fillMemory16(0xCDCD, 0x7000, "", 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> <p><i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 132.</p> <p><i>length</i> An integer that defines how many 4-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	<code>int 0</code>								
For use with	All C-SPY drivers.								
Description	Fills a specified memory area with a 4-byte value.								

Example

```
__fillMemory32(0x0000FFFF, 0x4000, "", 0x1000, "XOR");
```

__isBatchMode

Syntax

```
__isBatchMode()
```

Return value

Result	Value
True	int 1
False	int 0

Table 15: __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*.

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 16: __loadImage return values

For use with	All C-SPY drivers.
Description	<p>Loads an image (debug file).</p> <p>Note: For some products, images are only downloaded to RAM and no flash loading will be performed.</p>
Example 1	<p>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 <code>execUserSetup</code> macro in a C-SPY macro file, which you associate with your project:</p> <pre>__loadImage(ROMfile, 0x8000, 1);</pre> <p>This macro call loads the debug information for the ROM library <code>ROMfile</code> without downloading its contents (because it is presumably already in ROM). Then you can debug your application together with the library.</p>
Example 2	<p>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 <code>execUserSetup</code> macro in a C-SPY macro file, which you associate with your project:</p> <pre>__loadImage(ApplicationFile, 0x8000, 0);</pre> <p>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.</p>
See also	<i>Images</i> , page 308 and <i>Loading multiple images</i> , page 41.

__memoryRestore

Syntax	<code>__memoryRestore(zone, filename)</code>
Parameters	<p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 132.</p> <p><i>filename</i></p> <p>A string that specifies the file to be read. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide</i>.</p>
Return value	<code>int 0</code>

For use with	All C-SPY drivers.
Description	Reads the contents of a file and saves it to the specified memory zone.
Example	<code>__memoryRestore("", "c:\\temp\\saved_memory.hex");</code>
See also	<i>Memory Restore dialog box</i> , page 142.

__memorySave

Syntax	<code>__memorySave(start, stop, format, filename)</code>
Parameters	<p><i>start</i></p> <p>A string that specifies the first location of the memory area to be saved.</p> <p><i>stop</i></p> <p>A string that specifies the last location of the memory area to be saved.</p> <p><i>format</i></p> <p>A string that specifies the format to be used for the saved memory. Choose between:</p> <p><code>intel-extended</code></p> <p><code>motorola</code></p> <p><code>motorola-s19</code></p> <p><code>motorola-s28</code></p> <p><code>motorola-s37.</code></p> <p><i>filename</i></p> <p>A string that specifies the file to write to. The filename must include a path, which must either be absolute or use argument variables. For information about argument variables, see the <i>IDE Project Management and Building Guide</i>.</p>
Return value	<code>int 0</code>
For use with	All C-SPY drivers.
Description	Saves the contents of a specified memory area to a file.
Example	<code>__memorySave(":0x00", ":0xFF", "intel-extended", "c:\\temp\\saved_memory.hex");</code>

See also *Memory Save dialog box*, page 141.

__messageBoxYesCancel

Syntax `__messageBoxYesCancel(string message, string caption)`

Parameters

message
A message that will appear in the message box.

caption
The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 17: *__messageBoxYesCancel* return values

For use with All C-SPY drivers.

Description Displays a Yes/Cancel dialog box when called and returns the user input. Typically, this is useful for creating macros that require user interaction.

__messageBoxYesNo

Syntax `__messageBoxYesNo(string message, string caption)`

Parameters

message
A message that will appear in the message box.

caption
The title that will appear in the message box.

Return value

Result	Value
Yes	1
No	0

Table 18: *__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*.

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 19: __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*.

__orderInterrupt

Syntax `__orderInterrupt(specification, first_activation, repeat_interval, variance, infinite_hold_time, hold_time, probability)`

Parameters

specification

The interrupt (string). The specification can either be the full specification used in the device description file (*ddf*) or only the name. In the latter case the interrupt system will automatically get the description from the device description file.

first_activation

The first activation time in cycles (integer)

repeat_interval

The periodicity in cycles (integer)

variance

The timing variation range in percent (integer between 0 and 100)

infinite_hold_time

1 if infinite, otherwise 0.

hold_time

The hold time (integer)

probability

The probability in percent (integer between 0 and 100)

Return value

The macro returns an interrupt identifier (unsigned long).

If the syntax of *specification* is incorrect, it returns -1.

For use with	The C-SPY Simulator.
Description	Generates an interrupt.
Example	This example shows how you can use this macro to generate an interrupt: <pre>__orderInterrupt("USARTR_VECTOR", 4000, 2000, 0, 1, 0, 100);</pre>

__popSimulatorInterruptExecutingStack

Syntax	<code>__popSimulatorInterruptExecutingStack(void)</code>
Return value	<code>int 0</code>
For use with	The C-SPY Simulator.
Description	<p>Informs the interrupt simulation system that an interrupt handler has finished executing, as if the normal instruction used for returning from an interrupt handler was executed.</p> <p>This is useful if you are using interrupts in such a way that the normal instruction for returning from an interrupt handler is not used, for example in an operating system with task-switching. In this case, the interrupt simulation system cannot automatically detect that the interrupt has finished executing.</p>
See also	<i>Simulating an interrupt in a multi-task system</i> , page 217.

__readFile

Syntax	<code>__readFile(fileHandle, valuePtr)</code>
Parameters	<p><i>fileHandle</i> A macro variable used as filehandle by the <code>__openFile</code> macro.</p> <p><i>valuePtr</i> A pointer to a variable.</p>

Return value

Result	Value
Successful	0
Unsuccessful	Non-zero error number

Table 20: `__readFile` return values

For use with	All C-SPY drivers.
Description	<p>Reads a sequence of hexadecimal digits from the given file and converts them to an unsigned long which is assigned to the <i>value</i> parameter, which should be a pointer to a macro variable.</p> <p>Only printable characters representing hexadecimal digits and white-space characters are accepted, no other characters are allowed.</p>
Example	<pre>__var number; if (__readFile(myFileHandle, &number) == 0) { // Do something with number }</pre> <p>In this example, if the file pointed to by <code>myFileHandle</code> contains the ASCII characters 1234 abcd 90ef, consecutive reads will assign the values 0x1234 0xabcd 0x90ef to the variable <code>number</code>.</p>

__readFileByte

Syntax	<code>__readFileByte(<i>fileHandle</i>)</code>
Parameters	<p><i>fileHandle</i></p> <p>A macro variable used as filehandle by the <code>__openFile</code> macro.</p>
Return value	-1 upon error or end-of-file, otherwise a value between 0 and 255.
For use with	All C-SPY drivers.
Description	Reads one byte from a file.
Example	<pre>__var byte; while ((byte = __readFileByte(myFileHandle)) != -1) { /* Do something with byte */ }</pre>

__readMemory8, __readMemoryByte

Syntax	<pre>__readMemory8(<i>address</i>, <i>zone</i>) __readMemoryByte(<i>address</i>, <i>zone</i>)</pre>
--------	---

Parameters	<p><i>address</i></p> <p>The memory address (integer).</p> <p><i>zone</i></p> <p>A string that specifies the memory zone, see <i>C-SPY memory zones</i>, page 132.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads one byte from a given memory location.
Example	<code>__readMemory8(0x0108, "");</code>

__readMemory16

Syntax	<code>__readMemory16(<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 132.</p>
Return value	The macro returns the value from memory.
For use with	All C-SPY drivers.
Description	Reads a two-byte word from a given memory location.
Example	<code>__readMemory16(0x0108, "");</code>

__readMemory32

Syntax	<code>__readMemory32(<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 132.</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, "");</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</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 235.

__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 127.</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 21: __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	See <i>Supported debugger features</i> in the release notes for information about applicability.						
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 107.						

__setDataBreak

Syntax	<code>__setDataBreak(<i>location</i>, <i>count</i>, <i>condition</i>, <i>cond_type</i>, <i>access</i>, <i>action</i>)</code>						
Parameters	<p><i>location</i></p> <p>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 25.</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>access</i></p> <p>The memory access type: "R", for read, "W" for write, or "RW" for read/write.</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 22: __setDataBreak 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	See <i>Supported debugger features</i> in the release notes for information about applicability.						
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(":0x4710", 3, "d>6", "TRUE",
    "W", "ActionData()");
...
__clearBreak(brk);
```

See also

Breakpoints, page 107.

__setDataLogBreak

Syntax

```
__setDataLogBreak(variable, access)
```

Parameters

variable

A string that defines the variable the breakpoint is set on, a variable of integer type with static storage duration. The microcontroller must also be able to access the variable with a single-instruction memory access, which means that you can only set data log breakpoints on variables up to a certain size.

access

The memory access type: "R", for read, "W" for write, or "RW" for read/write.

Return value

Result	Value
Successful	An unsigned integer uniquely identifying the breakpoint. This value must be used to clear the breakpoint.
Unsuccessful	0

Table 23: __setDataLogBreak return values

For use with

See *Supported debugger features* in the release notes for information about applicability.

Description

Sets a data log breakpoint, that is, a breakpoint which is triggered when a specified variable is accessed. Note that a data log breakpoint does not stop the execution, it just generates a data log.

Example

```
__var brk;
brk = __setDataLogBreak("MyVar", "R");
...
__clearBreak(brk);
```

See also

Breakpoints, page 107 and *Getting started using data logging*, page 77.

__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 25.

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 24: __setLogBreak return values

For use with

See *Supported debugger features* in the release notes for information about applicability.

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 243 and *Breakpoints*, page 107.

__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 127.

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 25: *__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 127.

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 26: __setTraceStartBreak return values

For use with See *Supported debugger features* in the release notes for information about applicability.

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 107.

__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 127.

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 27: __setTraceStopBreak return values

For use with

See *Supported debugger features* in the release notes for information about applicability.

Description

Sets a breakpoint at the specified location. When that breakpoint is triggered, the trace system is stopped.

Example

See *__setTraceStartBreak*, page 272.

See also *Breakpoints*, page 107.

__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 28: *__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 241.

__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 241.

__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 241.

__toString

Syntax `__toString(C_string, maxlength)`

Parameters *C_string*

Any null-terminated C string.

maxlength

The maximum length of the returned macro string.

Return value Macro string.

For use with All C-SPY drivers.

Description This macro is used for converting C strings (`char*` or `char[]`) into macro strings.

Example Assuming your application contains this definition:

```
char const * hpstr = "Hello World!";
```

this macro call:

```
__toString(hptr, 5)
```

would return the macro string containing `Hello`.

See also *Macro strings*, page 241.

__ToUpper

Syntax `__ToUpper(macroString)`

Parameters *macroString*
A macro string.

Return value The converted string.

For use with All C-SPY drivers.

Description This macro returns a copy of the parameter *macroString* where all the characters have been converted to upper case.

Example `__ToUpper("string")`
The resulting macro string contains `STRING`.

See also *Macro strings*, page 241.

__unloadImage

Syntax `__unloadImage(module_id)`

Parameters *module_id*
An integer which represents a unique module identification, which is retrieved as a return value from the corresponding `__loadImage` C-SPY macro.

Return value

Value	Result
<i>module_id</i>	A unique module identification (the same as the input parameter).
int 0	The unloading failed.

Table 29: *__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 41 and *Images*, page 308.**__writeFile**

Syntax

`__writeFile(fileHandle, value)`

Parameters

*fileHandle*A macro variable used as filehandle by the `__openFile` macro.*value*

An integer.

Return value

int 0

For use with

All C-SPY drivers.

Description

Prints the integer value in hexadecimal format (with a trailing space) to the file *file*.**Note:** The `__fmessage` statement can do the same thing. The `__writeFile` macro is provided for symmetry with `__readFile`.**__writeFileByte**

Syntax

`__writeFileByte(fileHandle, value)`

Parameters

*fileHandle*A macro variable used as filehandle by the `__openFile` macro.*value*

An integer.

Return value	int 0
For use with	All C-SPY drivers.
Description	Writes one byte to the file <i>fileHandle</i> .

__writeMemory8, __writeMemoryByte

Syntax	<code>__writeMemory8(value, address, zone)</code> <code>__writeMemoryByte(value, address, zone)</code>
Parameters	<i>value</i> An integer. <i>address</i> The memory address (integer). <i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 132.
Return value	int 0
For use with	All C-SPY drivers.
Description	Writes one byte to a given memory location.
Example	<code>__writeMemory8(0x2F, 0x8020, "");</code>

__writeMemory16

Syntax	<code>__writeMemory16(value, address, zone)</code>
Parameters	<i>value</i> An integer. <i>address</i> The memory address (integer). <i>zone</i> A string that specifies the memory zone, see <i>C-SPY memory zones</i> , page 132.
Return value	int 0

For use with	All C-SPY drivers.
Description	Writes two bytes to a given memory location.
Example	<code>__writeMemory16(0x2FFF, 0x8020, "");</code>

__writeMemory32

Syntax	<code>__writeMemory32(value, address, zone)</code>
Parameters	<p><i>value</i> 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 132.</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, "");</code>

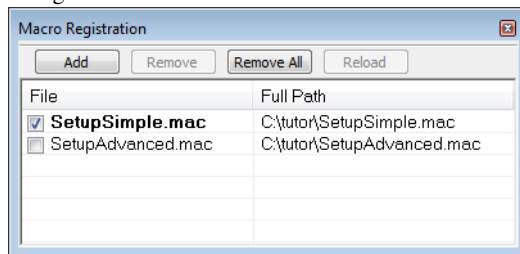
Graphical environment for macros

Reference information about:

- *Macro Registration window*, page 281
- *Debugger Macros window*, page 283
- *Macro Quicklaunch window*, page 285

Macro Registration window

The Macro Registration window is available from the **View>Macros** submenu during a debug session.



Use this window to list, register, and edit your debugger macro files.

Double-click a macro file to open it in the editor window and edit it.

Requirements

None; this window is always available.

Display area

This area contains these columns:

File

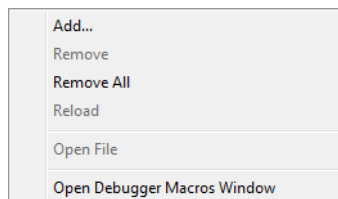
The name of an available macro file. To register the macro file, select the check box to the left of the filename. The name of a registered macro file appears in bold style.

Full path

The path to the location of the added macro file.

Context menu

This context menu is available:



These commands are available:

Add

Opens a file browser where you can locate the macro file that you want to add to the list. This menu command is also available as a function button at the top of the window.

Remove

Removes the selected debugger macro file from the list. This menu command is also available as a function button at the top of the window.

Remove All

Removes all macro files from the list. This menu command is also available as a function button at the top of the window.

Reload

Registers the selected macro file. Typically, this is useful when you have edited a macro file. This menu command is also available as a function button at the top of the window.

Open File

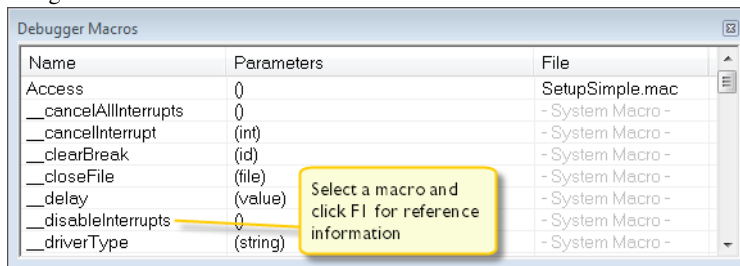
Opens the selected macro file in the editor window.

Open Debugger Macros Window

Opens the Debugger Macros window.

Debugger Macros window

The Debugger Macros window is available from the **View>Macro** submenu during a debug session.



Use this window to list all registered debugger macro functions, either predefined system macros or your own. This window is useful when you edit your own macro functions and want an overview of all available macros that you can use.

Double-clicking a macro defined in a file opens that file in the editor window.

To open a macro in the **Macro Quicklaunch** window, drag it from the **Debugger Macros** window and drop it in the **Macro Quicklaunch** window.

Select a macro and press F1 to get online help information for that macro.

Requirements

None; this window is always available.

Display area

This area contains these columns:

Name

The name of the debugger macro.

Parameters

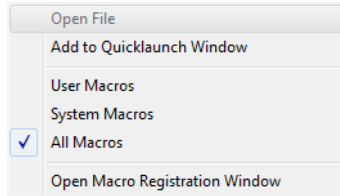
The parameters of the debugger macro.

File

For macros defined in a file, the name of the file is displayed. For predefined system macros, -System Macro- is displayed.

Context menu

This context menu is available:



These commands are available:

Open File

Opens the selected debugger macro file in the editor window.

Add to Quicklaunch Window

Adds the selected macro to the **Macro Quicklaunch** window.

User Macros

Lists only the debugger macros that you have defined yourself.

System Macros

Lists only the predefined system macros.

All Macros

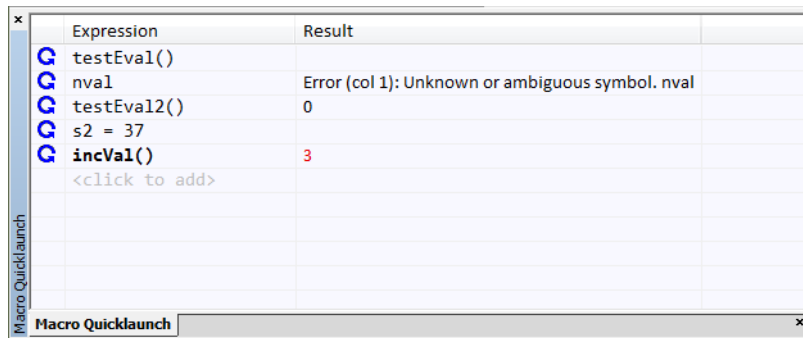
Lists all debugger macros, both predefined system macros and your own.

Open Macro Registration Window

Opens the **Macro Registration** window.

Macro Quicklaunch window

The **Macro Quicklaunch** window is available from the **View** menu.



Expression	Result
testEval()	
nval	Error (col 1): Unknown or ambiguous symbol. nval
testEval2()	0
s2 = 37	
incVal()	3
<click to add>	

Use this window to evaluate expressions, typically C-SPY macros.

For some devices, there are predefined C-SPY macros available with device support, typically provided by the chip manufacturer. These macros are useful for performing certain device-specific tasks. The macros are available in the **Macro Quicklaunch** window and are easily identified by their green icon,

The **Macro Quicklaunch** window is similar to the **Quick Watch** window, but is primarily designed for evaluating C-SPY macros. The window gives you precise control over when to evaluate an expression.

To add an expression:

- I Choose one of these alternatives:
 - Drag the expression to the window
 - In the **Expression** column, type the expression you want to examine.

If the expression you add and want to evaluate is a C-SPY macro, the macro must first be registered, see *Using C-SPY macros*, page 235.

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

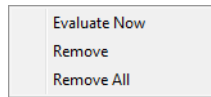
One or several expressions that you want to evaluate. Click <click to add> to add an expression. If the return value has changed since last time, the value will be displayed in red.

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 296.
- 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>cpuname</i> \bin.
<i>driver_DLL</i>	The C-SPY driver DLL file; available in <i>cpuname</i> \bin.
<i>debug_file</i>	The object file that you want to debug (filename extension <i>dxx</i>). See also <i>--debugfile</i> , page 292.
<i>cspybat_options</i>	The command line options that you want to pass to <code>cspybat</code> . Note that these options are optional. For information about each option, see <i>Reference information on C-SPY command line options</i> , page 290.

Table 30: *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 290.

Table 30: 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

This section gives an overview of each `cspybat` option and each generic option available to the C-SPY drivers. For information about driver-specific options, see *Supported debugger features* in the release notes.

GENERAL CSPYBAT OPTIONS

<code>--backend</code>	Marks the beginning of the parameters to be sent to the C-SPY driver (mandatory).
<code>--code_coverage_file</code>	Enables the generation of code coverage information and places it in a specified file.
<code>--cycles</code>	Specifies the maximum number of cycles to run.
<code>--debugfile</code>	Specifies an alternative debug file.
<code>--download_only</code>	Downloads a code image without starting a debug session afterwards.
<code>-f</code>	Extends the command line.
<code>--flash_loader</code>	Specifies a flash loader specification XML file.
<code>--leave_running</code>	Starts the execution on the target and then exits but leaves the target running.
<code>--macro</code>	Specifies a macro file to be used.

<code>--macro_param</code>	Assigns a value to a C-SPY macro parameter.
<code>--plugin</code>	Specifies a plugin file to be used.
<code>--silent</code>	Omits the sign-on message.
<code>--timeout</code>	Limits the maximum allowed execution time.

OPTIONS AVAILABLE FOR ALL C-SPY DRIVERS

<code>-p</code>	Specifies the device description file to be used.
-----------------	---

OPTIONS AVAILABLE FOR THE SIMULATOR DRIVER

<code>--disable_interrupts</code>	Disables the interrupt simulation.
<code>--mapu</code>	Activates memory access checking.

Reference information on C-SPY command line options

This section gives detailed reference information about each `cspybat` option and each option available to the C-SPY drivers.


--backend

Syntax	<code>--backend {<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 code coverage information. The code coverage information will be generated after the execution has completed and you can find it in the specified file. 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 205.
	 To set this option, choose View>Code Coverage , right-click and choose Save As when the C-SPY debugger is running.


--cycles

Syntax	<code>--cycles cycles</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>cycles</i> The number of cycles to run.
For use with	<code>cspybat</code>
Description	Use this option to specify the maximum number of cycles to run. If the target program executes longer than the number of cycles specified, the target program will be aborted. Using this option requires that the C-SPY driver you are using supports a cycle counter, and that it can be sampled while executing.  This option is not available in the IDE.


--debugfile

Syntax	<code>--debugfile filename</code>
Parameters	<i>filename</i> The name of the debug file to use.
For use with	<code>cspybat</code> This option can be placed both before and after the <code>--backend</code> option on the command line.
Description	Use this option to make <code>cspybat</code> use the specified debug file instead of the one used in the generated <code>cpsybat.bat</code> file.  This option is not available in the IDE.


--disable_interrupts

Syntax	<code>--disable_interrupts</code>
For use with	The C-SPY Simulator driver.
Description	Use this option to disable the interrupt simulation.  To set this option, choose Simulator>Interrupt Setup and deselect the Enable interrupt simulation option.


--download_only

Syntax	<code>--download_only</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
For use with	<code>cspybat</code>
Description	Use this option to download the code image without starting a debug session afterwards.  o set a related option, choose: Project>Options>Debugger>Setup and deselect Run to .


-f

Syntax	<code>-f filename</code>
Parameters	<i>filename</i> A text file that contains the commands (default filename extension <code>.xml</code>).
For use with	<code>cspybat</code> This option can be placed both before and after the <code>--backend</code> option on the command line.
Description	Use this option to make <code>cspybat</code> read command line options from the named file. In the command file, you format the items exactly as if they were on the command line itself, except that you may use multiple lines, because the newline character is treated like a space or tab character. Both C and C++ style comments are allowed in the file. Double quotes behave in the same way as in the Microsoft Windows command line environment.
	 To set this option, use Project>Options>Debugger>Extra Options .


--flash_loader

Syntax	<code>--flash_loader filename</code> Note that this option must be placed before the <code>--backend</code> option on the command line.
Parameters	<i>filename</i> The flash loader specification XML file, with the filename extension <code>.board</code> .
For use with	<code>cspybat</code>
Description	Use this option to specify a flash loader specification xml file which contains all relevant information about the flash loading. There can be more than one such argument, in which case each argument will be processed in the specified order, resulting in several flash programming passes.
See also	The <i>IAR Embedded Workbench flash loader User Guide</i> .
	 To set related options, choose: Project>Options>Debugger>Download>Use flash loader(s)

--leave_running

Syntax	<code>--leave_running</code>
	Note that this option must be placed before the <code>--backend</code> option on the command line.
For use with	<code>cspybat</code>
Description	Makes <code>cspybat</code> start the execution on the target and then exits but leaves the target running.
	 To set this option, use Project>Options>Debugger>Extra Options

--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 234.
	 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 an `av` value to a C-SPY macro parameter. This option can be used more than once on the command line.

See also *Macro parameters*, page 241.



Project>Options>Debugger>Extra Options

--mapu

Syntax `--mapu`

For use with The C-SPY simulator driver.

Description Specify this option to use the segment information in the debug file for memory access checking. During the execution, the simulator will then check for accesses to unspecified memory ranges. If any such access is found, the C function call stack and a message will be printed on `stderr` and the execution will stop.

See also *Memory access checking*, page 133.



To set related options, choose:

Simulator>Memory Access Setup

-p

Syntax `-p filename`

Parameters *filename*

The device description file to be used.

For use with All C-SPY drivers.

Description Use this option to specify the device description file to be used.

See also *Selecting a device description file*, page 39.



Project>Options>Debugger>Setup>Device description file

--plugin

Syntax `--plugin filename`

Note that this option must be placed before the `--backend` option on the command line.

Parameters `filename`

The plugin file to be used (filename extension `.dll`).

For use with `cspybat`

Description Certain C/C++ standard library functions, for example `printf`, can be supported by C-SPY—for example, the C-SPY **Terminal I/O** window—instead of by real hardware devices. To enable such support in `cspybat`, a dedicated plugin module called `cpunamebat.dll` located in the `cpuname\bin` directory must be used.

Use this option to include this plugin during the debug session. This option can be used more than once on the command line.

Note: You can use this option to include also other plugin modules, but in that case the module must be able to work with `cspybat` specifically. This means that the C-SPY plugin modules located in the `common\plugin` directory cannot normally be used with `cspybat`.



Project>Options>Debugger>Plugins

--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	<i>milliseconds</i> The number of milliseconds before the execution stops.
For use with	cspybat
Description	Use this option to limit the maximum allowed execution time.  This option is not available in the IDE.

Flash loaders

- Introduction to the flash loader
- Using flash loaders
- Reference information on the flash loader

Note that flash loaders might not be supported by the IAR Embedded Workbench version you are using. See *Supported debugger features* in the release notes.

Introduction to the flash loader

A flash loader is an agent that is downloaded to the target. It fetches your application from the debugger and programs it into flash memory. The flash loader uses the file I/O mechanism to read the application program from the host. You can select one or several flash loaders, where each flash loader loads a selected part of your application. This means that you can use different flash loaders for loading different parts of your application.

Flash loaders for various microcontrollers is provided with some IAR Embedded Workbench versions. The flash loader API, documentation, and several implementation examples are available to make it possible for you to implement your own flash loader.

Using flash loaders

These tasks are covered:

- Setting up the flash loader(s)
- The flash loading mechanism
- Build considerations
- Aborting a flash loader.

SETTING UP THE FLASH LOADER(S)

To use a flash loader for downloading your application:

- 1** Choose **Project>Options**.
- 2** Choose the **Debugger** category and click the **Download** tab.

- 3 Select the **Use Flash loader(s)** option. A default flash loader configured for the device you have specified will be used. The configuration is specified in a preconfigured board file.
- 4 To override the default flash loader or to modify the behavior of the default flash loader to suit your board, select the **Override default. board file** option, and **Edit** to open the **Flash Loader Configuration** dialog box. A copy of the *.board file will be created in your project directory and the path to the *.board file will be updated accordingly.
- 5 The **Flash Loader Overview** dialog box lists all currently configured flash loaders, see *Flash Loader Overview dialog box*, page 299. You can either select a flash loader or open the **Flash Loader Configuration** dialog box.

In the **Flash Loader Configuration** dialog box, you can configure the download. For more information about the various flash loader options, see *Flash Loader Configuration dialog box*, page 301.

THE FLASH LOADING MECHANISM

When the **Use flash loader(s)** option is selected and one or several flash loaders have been configured, these steps are performed when the debug session starts.

Steps 1 to 4 are performed for each flash loader in the flash loader configuration.

- 1 C-SPY downloads the flash loader into target RAM.
- Steps 2 to 4 are performed one or more times depending on the size of the RAM and the size of the application image.
- 2 C-SPY writes code/data from the application image into target RAM (RAM buffer).
 - 3 C-SPY starts execution of the flash loader.
 - 4 The flash loader reads data from the RAM buffer and programs the flash memory.
 - 5 The application image now resides in flash memory and can be started. The flash loader and the RAM buffer are no longer needed, so RAM is fully available to the application in the flash memory.

BUILD CONSIDERATIONS

This text is only applicable if you are using the XLINK linker.

When you build an application that will be downloaded to flash, special consideration is needed. Two output files must be generated. The first is the usual UBROF file (dxx) that provides the debugger with debug and symbol information. The second file is a simple-code file (filename extension sim) that will be opened and read by the flash loader when it downloads the application to flash memory.

The simple-code file must have the same path and name as the UBROF file except for the filename extension.

ABORTING A FLASH LOADER

To abort a flash loader:

- 1 Press Ctrl+Shift- (minus) for a short while.
- 2 A message that says that the flash loader has aborted is displayed in the Debug Log window.

This method can be used if you suspect that something is wrong with the execution, for example because it seems not to terminate in a reasonable time.

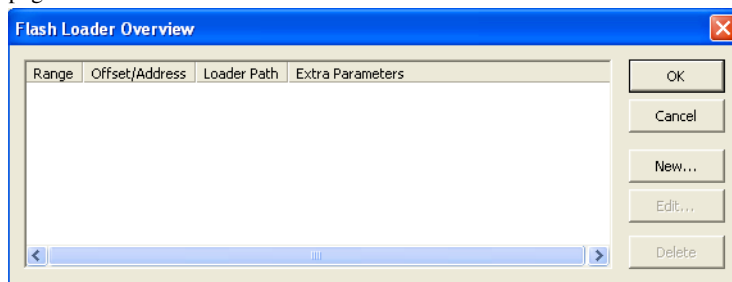
Reference information on the flash loader

Reference information about:

- *Flash Loader Overview dialog box*, page 299
- *Flash Loader Configuration dialog box*, page 301.

Flash Loader Overview dialog box

The **Flash Loader Overview** dialog box is available from the **Debugger>Download** page.



This dialog box lists all defined flash loaders. If you have selected a device on the **General Options>Target** page for which there is a flash loader, this flash loader is by default listed in the **Flash Loader Overview** dialog box.

The display area

Each row in the display area shows how you have set up one flash loader for flashing a specific part of memory:

Range

The part of your application to be programmed by the selected flash loader.

Offset/Address

The start of the memory where your application will be flashed. If the address is preceded with a, the address is absolute. Otherwise, it is a relative offset to the start of the memory.

Loader Path

The path to the flash loader *.flash file to be used (*.out for old-style flash loaders).

Extra Parameters

List of extra parameters that will be passed to the flash loader.

Click on the column headers to sort the list by range, offset/address, etc.

Function buttons

These function buttons are available:

OK

The selected flash loader(s) will be used for downloading your application to memory.

Cancel

Standard cancel.

New

Displays a dialog box where you can specify what flash loader to use, see *Flash Loader Configuration dialog box*, page 301.

Edit

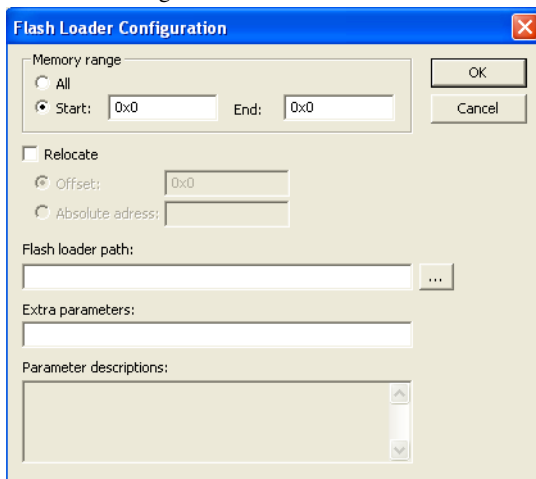
Displays a dialog box where you can modify the settings for the selected flash loader, see *Flash Loader Configuration dialog box*, page 301.

Delete

Deletes the selected flash loader configuration.

Flash Loader Configuration dialog box

The **Flash Loader Configuration** dialog box is available from the **Flash Loader Overview** dialog box.



Use the **Flash Loader Configuration** dialog box to configure the download to suit your board. A copy of the default `board` file will be created in your project directory.

Memory range

Specify the part of your application to be downloaded to flash memory. Choose between:

All

The whole application is downloaded using this flash loader.

Start/End

Specify the start and the end of the memory area for which part of the application will be downloaded.

Relocate

Overrides the default flash base address, in other words, relocates the location of the application in memory. This means that you can flash your application to a different location from where it was linked. Choose between:

Offset

A numeric value for a relative offset. This offset will be added to the addresses in the application file.

Absolute address

A numeric value for an absolute base address where the application will be flashed. The lowest address in the application will be placed on this address. Note that you can only use one flash loader for your application when you specify an absolute address.

You can use these numeric formats:

- 123456, decimal numbers
- 0x123456, hexadecimal numbers
- 0123456, octal numbers

The default base address used for writing the first byte—the lowest address—to flash is specified in the linker configuration file used for your application. However, it can sometimes be necessary to override the flash base address and start at a different location in the address space. This can, for example, be necessary for devices that remap the location of the flash memory.

Flash loader path

Use the text box to specify the path to the flash loader file (`*.flash`) to be used by your board configuration.

Extra parameters

Some flash loaders define their own set of specific options. Use this text box to specify options to control the flash loader. For information about available flash loader options, see the **Parameter descriptions** field.

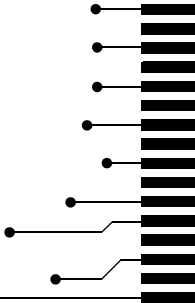
Parameter descriptions

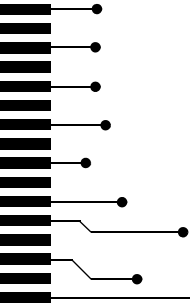
Displays a description of the extra parameters specified in the **Extra parameters** text box.

Part 4. Additional reference information

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

- Debugger options
- Additional information on C-SPY drivers





Debugger options

- Setting debugger options
- Reference information on general debugger 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 305.

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.

5 To restore all settings to the default factory settings, click the **Factory Settings** button.

6 When you have set all the required options, click **OK** in the **Options** dialog box.

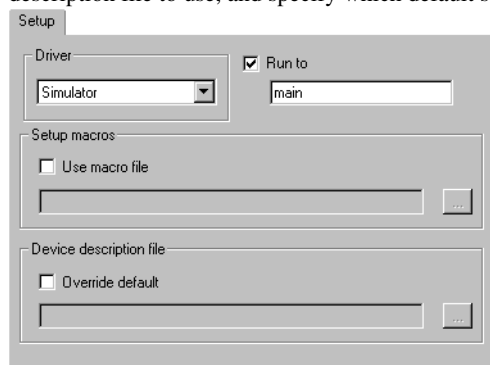
Reference information on general debugger options

Reference information about:

- Setup
- Download
- Images
- Extra Options
- Plugins

Setup

The general **Setup** options select the C-SPY driver, the setup macro file, and device description file to use, and specify which default source code location to run to.



Driver

Selects the C-SPY driver for the target system you have.

Run to

Specifies the location C-SPY runs to when the debugger starts after a reset. By default, C-SPY runs to the `main` function.

To override the default location, specify the name of a different location you want C-SPY to run to. You can specify assembler labels or whatever can be evaluated as such, for example function names.

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.

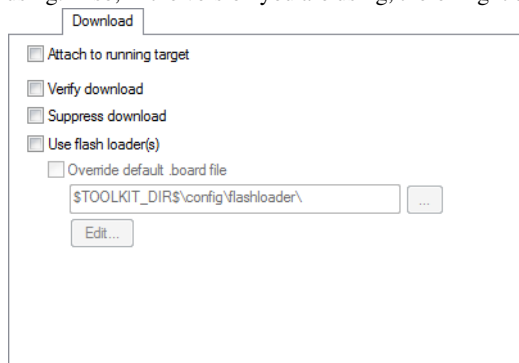
Device description file

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 42.

Download

By default, C-SPY downloads the application to RAM or flash when a debug session starts. The **Download** options let you modify the behavior of the download. Note that these options might not be available in the IAR Embedded Workbench version you are using. Also, in the version you are using, there might be additional options on this page.



Attach to running target

Makes the debugger attach to a running application at its current location, without resetting the target system. To avoid unexpected behavior when using this option, the **Debugger>Setup** option **Run to** should be deselected.

Verify download

Verifies that the downloaded code image can be read back from target memory with the correct contents.

Suppress download

Disables the downloading of code, while preserving the present content of the flash. This command is useful if you want to debug an application that already resides in target memory.

If this option is combined with the **Verify download** option, the debugger will read back the code image from non-volatile memory and verify that it is identical to the debugged application.

Use flash loader(s)

Use this option to use one or several flash loaders for downloading your application to flash memory. If a flash loader is available for the selected chip, it is used by default. Press the **Edit** button to display the **Flash Loader Overview** dialog box.

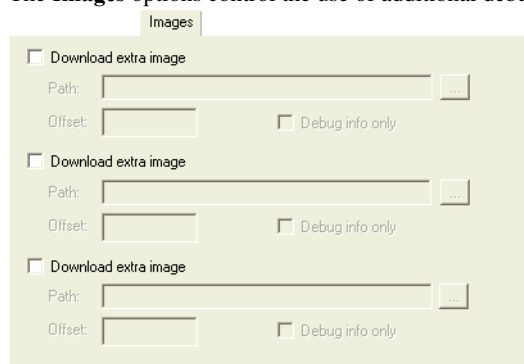
For more information about flash loaders, see *Flash loaders*, page 297.

Override default .board file

A default flash loader is selected based on your choice of device on the **General Optios>Target** page. To override the default flash loader, select **Override default .board file** and specify the path to the flash loader you want to use. A browse button is available for your convenience. Click **Edit** to display the **Flash Loader Overview** dialog box. For more information, see *Flash Loader Overview dialog box*, page 299.

Images

The **Images** options control the use of additional debug files to be downloaded.



Note: For some products, images are only downloaded to RAM and no flash loading will be performed.

Download extra Images

Controls the use of additional debug files to be downloaded:

Path

Specify the debug file to be downloaded. A browse button is available for your convenience.

Offset

Specify an integer that determines the destination address for the downloaded debug file.

Debug info only

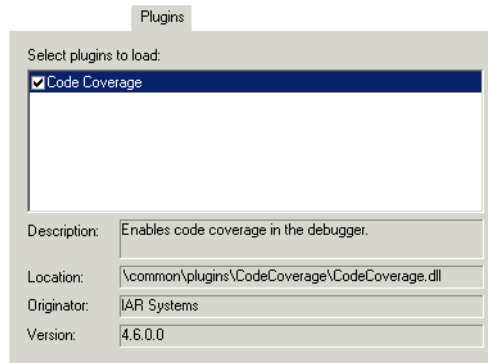
Makes the debugger download only debug information, and not the complete debug file.

If you want to download more than three images, use the related C-SPY macro, see *__loadImage*, page 257.

For more information, see *Loading multiple images*, page 41.

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 `cpuname\plugins` directory.

Originator

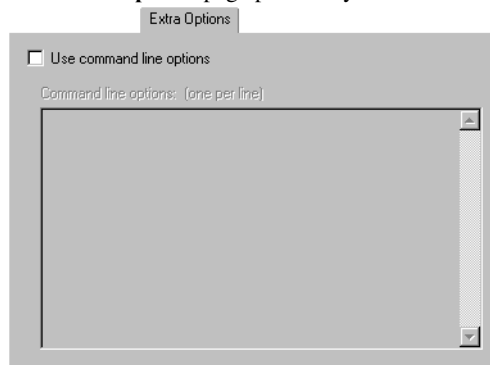
Informs about the originator of the plugin module, which can be modules provided by IAR Systems or by third-party vendors.

Version

Informs about the version number.

Extra Options

The **Extra Options** page provides you with a command line interface to C-SPY.



Use command line options

Specify command line arguments that are not supported by the IDE to be passed to C-SPY.

Note that it is possible to use the `/args` option to pass command line arguments to the debugged application.

Syntax: `/args arg0 arg1 ...`

Multiple lines with `/args` are allowed, for example:

```
/args --logfile log.txt
```

```
/args --verbose
```

If you use `/args`, these variables must be defined in your application:

```
/* __argc, the number of arguments in __argv. */
__no_init int __argc;
```

```
/* __argv, an array of pointers to strings that holds the
arguments; must be large enough to fit the number of
parameters.*/
__no_init const char * __argv[MAX_ARGS];
```

```
/* __argvbuf, a storage area for __argv; must be large enough to
hold all command line parameters. */
__no_init __root char __argvbuf[MAX_ARG_SIZE];
```


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 311
- *Simulator menu*, page 312

C-SPY driver

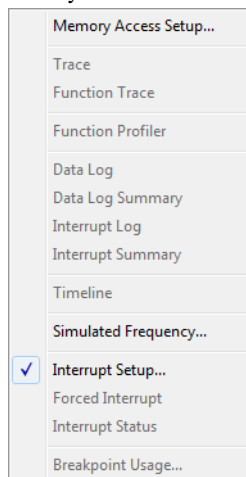
Before you start the C-SPY debugger, you must first specify a C-SPY driver in the **Options** dialog box, using the option **Debugger>Setup>Driver**.

When you start a debug session, a menu specific to that C-SPY driver will appear on the menu bar, with commands specific to the driver.

When we in this guide write “choose *C-SPY driver*>” followed by a menu command, *C-SPY driver* refers to the menu. If the feature is supported by the driver, the command will be on the menu.

Simulator menu

When you use the simulator driver, the **Simulator** menu is added to the menu bar.



Menu commands

These commands are available on the menu:

Memory Access Setup

Displays a dialog box to simulate memory access checking by specifying memory areas with different access types, see *Memory Access Setup dialog box*, page 165.

Memory Configuration

Displays a dialog box where you configure C-SPY to match the memory of your device, see *Memory Configuration dialog box, for the C-SPY simulator*, page 156.

Trace

Opens a window which displays the collected trace data, see *Trace window*, page 174.

Function Trace

Opens a window which displays the trace data for function calls and function returns, see *Function Trace window*, page 177.

Function Profiler

Opens a window which shows timing information for the functions, see *Function Profiler window*, page 199.

Data Log

Opens a window which logs accesses to up to four different memory locations or areas, see *Data Log window*, page 95.

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 97.

Interrupt Log

Opens a window which displays the status of all defined interrupts, see *Interrupt Log window*, page 225.

Interrupt Log Summary

Opens a window which displays a summary of the status of all defined interrupts, see *Interrupt Log Summary window*, page 228.

Timeline

Opens a window which gives a graphical view of various kinds of information on a timeline, see *Timeline window*, page 177.

Simulated Frequency

Opens the **Simulated Frequency** dialog box where you can specify the simulator frequency used when the simulator displays time information, for example in the log windows. Note that this does not affect the speed of the simulator.

Interrupt Setup

Displays a dialog box where you can configure C-SPY interrupt simulation, see *Interrupt Setup dialog box*, page 219.

Forced Interrupts

Opens a window from where you can instantly trigger an interrupt, see *Forced Interrupt window*, page 222.

Interrupt Status

Opens a window from where you can instantly trigger an interrupt, see *Interrupt Status window*, page 223.

Breakpoint Usage

Displays a window which lists all active breakpoints, see *Breakpoint Usage window*, page 118.

Reference information on the C-SPY simulator

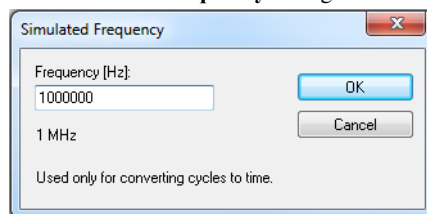
This section gives additional reference information the C-SPY simulator, reference information not provided elsewhere in this documentation.

Reference information about:

- *Simulated Frequency dialog box*, page 314

Simulated Frequency dialog box

The **Simulated Frequency** dialog box is available from the C-SPY driver menu.



Use this dialog box to specify the simulator frequency used when the simulator displays time information.

Requirements

The C-SPY simulator.

Frequency

Specify the frequency in Hz.

Resolving problems

These topics are covered:

- Write failure during load
- No contact with the target hardware
- Slow stepping speed

Debugging using the C-SPY hardware debugger systems requires interaction between many systems, independent from each other. For this reason, setting up this debug system can be a complex task. If something goes wrong, it might be difficult to locate the cause of the problem.

This section includes suggestions for resolving the most common problems that can occur when debugging with the C-SPY hardware debugger systems.

For problems concerning the operation of the evaluation board, refer to the documentation supplied with it, or contact your hardware distributor.

WRITE FAILURE DURING LOAD

There are several possible reasons for write failure during load. The most common is that your application has been incorrectly linked:

- Check the contents of your linker configuration file and make sure that your application has not been linked to the wrong address
- Check that you are using the correct linker configuration file.



In the IDE, the linker configuration file is automatically selected based on your choice of device.

To choose a device:

- 1 Choose **Project>Options**.
- 2 Select the **General Options** category.
- 3 Click the **Target** tab.
- 4 Choose the appropriate device from the **Device** drop-down list.

To override the default linker configuration file:

- 1 Choose **Project>Options**.
- 2 Select the **Linker** category.
- 3 Click the **Config** tab.
- 4 Choose the appropriate linker configuration file in the **Linker configuration file** area.

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.

SLOW STEPPING SPEED

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

- If you are using a hardware debugger system, keep track of how many hardware breakpoints that are used and make sure some of them are left for stepping.
Stepping in C-SPY is normally performed using breakpoints. When C-SPY performs a step command, a breakpoint is set on the next statement and the application executes until it reaches this breakpoint. If you are using a hardware debugger system, the number of hardware breakpoints—typically used for setting a stepping breakpoint in code that is located in flash/ROM memory—is limited. If you, for example, step into a `C switch` statement, breakpoints are set on each branch; this might consume several hardware breakpoints. If the number of available hardware breakpoints is exceeded, C-SPY switches into single stepping on assembly level, which can be very slow.
For more information, see *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 **Register** window. Displaying many SFR registers might slow down stepping because all registers must be read from the hardware after each step. See *Defining application-specific register groups*, page 135.
- Close the **Memory** and **Symbolic Memory** windows if they are open, because the visible memory must be read after each step and that might slow down stepping.
- Close any window that displays expressions such as **Watch**, **Live Watch**, **Locals**, **Statics** if it is open, because all these windows read memory after each step and that might slow down stepping.
- Close the **Stack** window if it is open. Choose **Tools>Options>Stack** and disable the **Enable graphical stack display and stack usage tracking** option if it is enabled.
- If possible, increase the communication speed between C-SPY and the target board/emulator.

A

Abort (Report Assert option)	70
absolute location, specifying for a breakpoint.	127
Access type (Edit Memory Access option)	160, 168
Access (Edit SFR option)	156
Add to Watch Window (Symbolic Memory window context menu)	146
Add (SFR Setup window context menu)	154
Address Range (Find in Trace option)	190
Address (Edit SFR option)	155
Ambiguous symbol (Resolve Symbol Ambiguity option)	94
application, built outside the IDE	40
assembler labels, viewing	76
assembler source code, fine-tuning	193
assembler symbols, using in C-SPY expressions	73
assembler variables, viewing.	76
assumptions, programming experience	19
Attach to program (debugger option)	309
Auto Scroll (Sampled Graphs window context menu)	104
Auto Scroll (Timeline window context menu)	182
Auto window	79
Autostep settings dialog box	70
Autostep (Debug menu)	47

B

--backend (C-SPY command line option)	290
backtrace information	
generated by compiler	58
viewing in Call Stack window	64
batch mode, using C-SPY in	287
Big Endian (Memory window context menu)	140
blocks, in C-SPY macros	243
bold style, in this guide	24
Break on Throw (Debug menu).	47
Break on Uncaught Exception (Debug menu).	47
Break (Debug menu).	46
breakpoint condition, example	114–115

Breakpoint Usage window	118
Breakpoint Usage (Simulator menu).	315
breakpoints	
code, example	267
connecting a C-SPY macro	238
consumers of	110
data	122
data log	124
description of	107
disabling used by Stack window	111
icons for in the IDE	110
in Memory window	113
listing all	118
profiling source	194, 200
reasons for using	107
setting	
in memory window	113
using system macros	114
using the dialog box	112
single-stepping if not available.	38
toggling	112
types of	108
useful tips.	114
Breakpoints dialog box	
Code	119
Data	122
Data Log	124
Immediate	126
Log	121
Trace Start	186
Trace Stop	187
Breakpoints window	117
Browse (Trace toolbar)	175
byte order, setting in Memory window	139

C

C function information, in C-SPY.	58
C symbols, using in C-SPY expressions	73

C variables, using in C-SPY expressions	72	current position, in C-SPY Disassembly window	62
Cache type (Edit Memory Range option)	164	cursor, in C-SPY Disassembly window	62
call chain, displaying in C-SPY	58	--cycles (C-SPY command line option)	291
Call stack information	58	C-SPY	
Call Stack window	64	batch mode, using in	287
for backtrace information	58	debugger systems, overview of	32
Call Stack (Timeline window context menu)	183	environment overview	29
__cancelAllInterrupts (C-SPY system macro)	250	plugin modules, loading	39
__cancelInterrupt (C-SPY system macro)	251	setting up	37–38
Clear All (Debug Log window context menu)	69	starting the debugger	39
Clear trace data (Trace toolbar)	175	C-SPY drivers	
Clear (Interrupt Log window context menu)	227, 230	specifying	308
__clearBreak (C-SPY system macro)	251	types of	33
CLIB		C-SPY expressions	72
consuming breakpoints	111	evaluating, using Macro Quicklaunch window	285
documentation	22	evaluating, using Quick Watch window	90
library reference information for	22	in C-SPY macros	242
naming convention	24	Tooltip watch, using	71
clock frequency, simulated	316	Watch window, using	71
__closeFile (C-SPY system macro)	251	C-SPY macros	
code breakpoints		blocks	243
overview	108	conditional statements	242
toggling	112	C-SPY expressions	242
Code Coverage window	206	examples	235
Code Coverage (Disassembly window context menu)	63	checking status of register	237
--code_coverage_file (C-SPY command line option)	291	creating a log macro	238
code, covering execution of	206	executing	235
command line options	290	connecting to a breakpoint	238
typographic convention	23	using Quick Watch	237
command prompt icon, in this guide	24	using setup macro and setup file	237
computer style, typographic convention	23	functions	74, 240
conditional statements, in C-SPY macros	242	loop statements	243
context menu, in windows	76	macro statements	242
conventions, used in this guide	23	parameters	241
Copy Window Contents (Disassembly window context menu)	64	setup macro file	234
Copy (Debug Log window context menu)	68	executing	237
copyright notice	2	setup macro functions	234
cspybat	287	summary	245
reading options from file (-f)	293	system macros, summary of	248

using	233
variables	74, 240
C-SPY options	
Extra Options	312
Images	310
Plugins	311
Setup	308
C-SPYLink	34
C-STAT for static analysis, documentation for	22
C++ terminology	23

D

data breakpoints, overview	108
Data Coverage (Memory window context menu)	140
data coverage, in Memory window	138
data log breakpoints, overview	109
Data Log Summary window	97
Data Log Summary (Simulator menu)	315
Data Log window	95
Data Log (Simulator menu)	315
Data Log (Timeline window context menu)	183
Data Sample Setup window	99
Data Sample window	100
Data Sample (Sampled Graphs window context menu)	104
ddf (filename extension), selecting a file	39
Debug Log window	68
Debug menu (C-SPY main window)	46
Debug (Report Assert option)	70
--debugfile (csybat option)	292
debugger concepts, definitions of	31
debugger drivers	
simulator	34
Debugger Macros window	283
debugger system overview	32
debugging projects	
externally built applications	40
loading multiple images	41
debugging, RTOS awareness	31

__delay (C-SPY system macro)	252
Delay (Autostep Settings option)	70
Delete (Breakpoints window context menu)	118
Delete (SFR Setup window context menu)	154
Delete/revert All Custom SFRs (SFR Setup window context menu)	154
Description (Edit Interrupt option)	221
description (interrupt property)	221
Device description file (debugger option)	308
device description files	39
definition of	42
memory zones	132
modifying	42
register zone	132
specifying interrupts	262
Disable All (Breakpoints window context menu)	118
Disable (Breakpoints window context menu)	118
__disableInterrupts (C-SPY system macro)	252
--disable_interrupts (C-SPY command line option)	292
Disassembly window	60
context menu	62
disclaimer	2
DLIB	
consuming breakpoints	111
documentation	22
naming convention	24
do (macro statement)	243
document conventions	23
documentation	
overview of guides	21
overview of this guide	20
this guide	19
--download_only (C-SPY command line option)	292
Driver (debugger option)	308
__driverType (C-SPY system macro)	252

E

Edit Expressions (Trace toolbar)	175
----------------------------------	-----

Edit Interrupt dialog box	220
Edit Memory Access dialog box	167
Edit Memory Range dialog box	155, 163
Edit Memory Range dialog box (C-SPY simulator)	159
Edit Settings (Trace toolbar)	175
Edit (Breakpoints window context menu)	117
Edit (SFR Setup window context menu)	154
edition, of this guide	2
Embedded C++ Technical Committee	23
Enable All (Breakpoints window context menu)	118
Enable interrupt simulation (Interrupt Setup option)	219
Enable Log File (Log File option)	69
Enable (Breakpoints window context menu)	118
Enable (Interrupt Log window context menu)	227, 230
Enable (Sampled Graphs window context menu)	104
Enable (Timeline window context menu)	183
__enableInterrupts (C-SPY system macro)	253
Enable/Disable Breakpoint (Call Stack window context menu)	66
Enable/Disable Breakpoint (Disassembly window context menu)	64
Enable/Disable (Trace toolbar)	175
End address (Edit Memory Access option)	168
endianness. <i>See</i> byte order	
Enter Location dialog box	127
__evaluate (C-SPY system macro)	253
Evaluate Now (Macro Quicklaunch window context menu)	286
examples	
C-SPY macros	235
interrupts	
interrupt logging	218
timer	216
macros	
checking status of register	237
creating a log macro	238
using Quick Watch	237
performing tasks and continue execution	115
tracing incorrect function arguments	114
execUserExecutionStarted (C-SPY setup macro)	246

execUserExecutionStopped (C-SPY setup macro)	246
execUserExit (C-SPY setup macro)	248
execUserFlashExit (C-SPY setup macro)	248
execUserFlashInit (C-SPY setup macro)	246
execUserFlashReset (C-SPY setup macro)	247
execUserPreload (C-SPY setup macro)	245
execUserPreReset (C-SPY setup macro)	247
execUserReset (C-SPY setup macro)	247
execUserSetup (C-SPY setup macro)	246
executed code, covering	206
execution history, tracing	174
expressions. <i>See</i> C-SPY expressions	
extended command line file, for cspybat	293
Extra Options, for C-SPY	312

F

-f (cspybat option)	293
Factory ranges (Memory Configuration option)	161
File format (Memory Save option)	141
file types	
device description, specifying in IDE	39
macro	38, 308
filename extensions	
ddf, selecting device description file	39
mac, using macro file	38
Filename (Memory Restore option)	142
Filename (Memory Save option)	142
Fill dialog box	143
__writeMemory8 (C-SPY system macro)	254
__writeMemory16 (C-SPY system macro)	255
__writeMemory32 (C-SPY system macro)	256
Find in Trace dialog box	189
Find in Trace window	190
Find (Memory window context menu)	140
Find (Trace toolbar)	175
first activation time (interrupt property)	
definition of	213
First activation (Edit Interrupt option)	221

flash loader

- parameters to control 304
- specifying the path to 304
- using 299

Flash Loader Overview dialog box 301

flash memory, load library module to 258

--flash_loader (C-SPY command line option) 293

for (macro statement) 243

Forced Interrupt window 222

Forced Interrupts (Simulator menu) 315

formats, C-SPY input 31

Function Profiler window 199

Function Profiler (Simulator menu) 314

Function Trace window 177

Function Trace (Simulator menu) 314

functions

- call stack information for 58
- C-SPY running to when starting 38, 308
- most time spent in, locating 193

G

Go to Source (Breakpoints window context menu) 117

Go to Source (Call Stack window context menu) 65

Go To Source (Timeline window context menu) 184

Go (Debug menu) 46, 57

Graphical bar (Memory Configuration dialog box) 162

H

highlighting, in C-SPY 58

Hold time (Edit Interrupt option) 221

hold time (interrupt property), definition of 213

I

icons, in this guide 24

if else (macro statement) 242

if (macro statement) 242

Ignore (Report Assert option) 70

illegal memory accesses, checking for 133

Images window 50

Images, loading multiple 310

immediate breakpoints, overview 109

Include (Log File option) 69

input formats, C-SPY 31

Input Mode dialog box 67

input, special characters in Terminal I/O window 67

installation directory 23

Instruction Profiling (Disassembly window context menu) . 63

Intel-extended, C-SPY input format 31

Intel-extended, C-SPY output format 33

Interrupt Log Summary window 228

Interrupt Log Summary (Simulator menu) 315

Interrupt Log window 225

Interrupt Log (Simulator menu) 315

Interrupt Setup dialog box 219

Interrupt Setup (Simulator menu) 315

Interrupt Status window 223

interrupt system, using device description file 215

Interrupt (Edit Interrupt option) 221

Interrupt (Timeline window context menu) 183

interrupts

- adapting C-SPY system for target hardware 215
- simulated, introduction to 211
- timer, example 216
- using system macros 214
- __isBatchMode (C-SPY system macro) 257

italic style, in this guide 23–24

I/O register. *See* SFR

L

labels (assembler), viewing 76

Length (Fill option) 143

library functions

- C-SPY support for using, plugin module 296
- online help for 22

lightbulb icon, in this guide	24
linker options	
typographic convention	23
consuming breakpoints	111
Little Endian (Memory window context menu)	139
Live Watch window	85
__loadImage (C-SPY system macro)	257
loading multiple debug files, list currently loaded.	50
loading multiple images	41
Locals window	81
log breakpoints, overview	108
Log File dialog box.	69
Logging>Set Log file (Debug menu)	48
Logging>Set Terminal I/O Log file (Debug menu).	48
loop statements, in C-SPY macros	243

M

mac (filename extension), using a macro file	38
--macro (C-SPY command line option)	294
macro files, specifying	38, 308
Macro Quicklaunch window	285
Macro Registration window	281
macro statements	242
macros	
executing	235
using	233
Macros (Debug menu)	48
--macro-param (C-SPY command line option)	294
main function, C-SPY running to when starting	38, 308
--mapu (C-SPY command line option)	295
memory access checking.	133
Memory access checking (Memory Access Setup option)	166
Memory access checking (Memory Configuration option)	158
Memory Access Setup dialog box.	165
Memory Access Setup (Simulator menu)	314
memory accesses, illegal.	133
Memory Configuration dialog box	160
Memory Configuration dialog box (in C-SPY simulator)	156

Memory Configuration (Simulator menu).	314
Memory Fill (Memory window context menu).	140
memory map.	165
Memory range (Edit Memory Access option).	168
Memory Restore dialog box	142
Memory Restore (Memory window context menu).	140
Memory Save dialog box	141
Memory Save (Memory window context menu)	140
Memory window.	137
memory zones.	132
in device description file	132
__memoryRestore (C-SPY system macro)	258
__memorySave (C-SPY system macro)	259
Memory>Restore (Debug menu)	47
Memory>Save (Debug menu).	47
menu bar, C-SPY-specific	45
__messageBoxYesCancel (C-SPY system macro)	260
__messageBoxYesNo (C-SPY system macro)	260
MISRA C	
documentation	22
Mixed Mode (Disassembly window context menu)	64
Motorola, C-SPY input format	31
Motorola, C-SPY output format	33
Move to PC (Disassembly window context menu)	62

N

Name (Edit SFR option).	155
naming conventions	24
Navigate (Sampled Graphs window context menu)	103
Navigate (Timeline window context menu)	182
New Breakpoint (Breakpoints window context menu)	118
Next Statement (Debug menu)	47
Next Symbol (Symbolic Memory window context menu)	146

O

__openFile (C-SPY system macro).	261
Operation (Fill option)	143

operators, sizeof in C-SPY 74
 optimizations, effects on variables 74
 options
 in the IDE 307
 on the command line 290, 312
 Options (Stack window context menu) 150
 __orderInterrupt (C-SPY system macro) 262
 Originator (debugger option) 311
 Override default .board file (debugger option) 310

P

-p (C-SPY command line option) 295
 parameters
 list of passed to the flash loader 302
 tracing incorrect values of 58
 typographic convention 23
 part number, of this guide 2
 peripheral units
 device-specific 42
 displayed in Register window 132
 in C-SPY expressions 73
 initializing using setup macros 234
 peripherals register. *See* SFR
 Please select one symbol
 (Resolve Symbol Ambiguity option) 94
 --plugin (C-SPY command line option) 296
 plugin modules (C-SPY) 34
 loading 39
 Plugins (C-SPY options) 311
 __popSimulatorInterruptExecutingStack (C-SPY
 system macro) 263
 pop-up menu. *See* context menu
 prerequisites, programming experience. 19
 Previous Symbol (Symbolic
 Memory window context menu) 146
 probability (interrupt property) 221
 definition of 213
 Probability % (Edit Interrupt option) 221
 Profile Selection (Timeline window context menu) 184

profiling
 analyzing data 195
 on function level 195
 on instruction level 197
 profiling information, on functions and instructions 193
 profiling sources
 breakpoints 194, 200
 sampling 194, 201
 trace (calls) 194, 200
 trace (flat) 194, 201
 program execution
 breaking 108
 in C-SPY 53
 programming experience. 19
 program. *See* application
 projects, for debugging externally built applications. 40
 publication date, of this guide 2

Q

Quick Watch window 90
 executing C-SPY macros 237

R

RAM (Edit Memory Access option) 164
 Range for (Viewing Range option) 185
 __readFile (C-SPY system macro) 263
 __readFileByte (C-SPY system macro) 264
 reading guidelines 19
 __readMemoryByte (C-SPY system macro) 264
 __readMemory8 (C-SPY system macro) 264
 __readMemory16 (C-SPY system macro) 265
 __readMemory32 (C-SPY system macro) 265
 reference information, typographic convention. 24
 Refresh (Debug menu) 48
 register groups 132
 predefined, enabling 151
 Register window 150

registered trademarks	2
__registerMacroFile (C-SPY system macro)	266
registers, displayed in Register window	150
Remove All (Macro Quicklaunch window context menu)	286
Remove (Macro Quicklaunch window context menu)	286
Repeat interval (Edit Interrupt option)	221
repeat interval (interrupt property), definition of	213
Replace (Memory window context menu)	140
Report Assert dialog box	70
Reset (Debug menu)	46
__resetFile (C-SPY system macro)	266
Resolve Source Ambiguity dialog box	128
Restore (macro statement)	243
ROM-monitor, definition of	33
ROM/Flash (Edit Memory Access option)	164
RTOS awareness debugging	31
RTOS awareness (C-SPY plugin module)	31
Run to Cursor (Call Stack window context menu)	65
Run to Cursor (Debug menu)	47
Run to Cursor (Disassembly window context menu)	63
Run to Cursor, command for executing	58
Run to (C-SPY option)	38, 308

S

Sampled Graphs window	102
sampling, profiling source	194, 201
Save Custom SFRs (SFR Setup window context menu)	155
Save to log file (context menu command)	227, 230
Save (Memory Save option)	142
Save (Trace toolbar)	175
Scale (Viewing Range option)	185
Select All (Debug Log window context menu)	68
Select Graphs (Sampled Graphs window context menu)	105
Select Graphs (Timeline window context menu)	184
Select plugins to load (debugger option)	311
Set Data Breakpoint (Memory window context menu)	140

Set Data Log	
Breakpoint (Memory window context menu)	141
Set Next Statement (Debug menu)	47
Set Next Statement (Disassembly window context menu)	64
__setCodeBreak (C-SPY system macro)	267
__setDataBreak (C-SPY system macro)	268
__setDataLogBreak (C-SPY system macro)	269
__setLogBreak (C-SPY system macro)	270
__setSimBreak (C-SPY system macro)	271
__setTraceStartBreak (C-SPY system macro)	272
__setTraceStopBreak (C-SPY system macro)	273
setup macro file, registering	38
setup macro functions	234
reserved names	245
Setup macros (debugger option)	308
Setup (C-SPY options)	308
setupsimple.mac	233
SFR	
in Register window	151
using as assembler symbols	73
SFR Setup window	152
SFR/Uncached (Edit Memory Access option)	165
shortcut menu. <i>See</i> context menu	
Show all images (Images window context menu)	50
Show All (SFR Setup window context menu)	154
Show Arguments (Call Stack window context menu)	65
Show Custom SFRs only (SFR Setup window context menu)	154
Show Cycles (Interrupt Log window context menu)	227, 230
Show Factory SFRs only (SFR Setup window context menu)	154
Show Numerical Value (Sampled Graphs window context menu)	105
Show Numerical Value (Timeline window context menu)	184
Show offsets (Stack window context menu)	149
Show only (Image window context menu)	50
Show Time (Interrupt Log window context menu)	227, 230
Show variables (Stack window context menu)	149
--silent (C-SPY command line option)	296
Simulated Frequency dialog box	316

simulating interrupts, enabling/disabling 219
 Simulator menu 314
 simulator, introduction 34
 Size (Edit SFR option) 156
 Size (Sampled Graphs window context menu) 105
 Size (Timeline window context menu) 184
 sizeof 74
 Solid Graph (Sampled Graphs window context menu) 105
 Solid Graph (Timeline window context menu) 183
 __sourcePosition (C-SPY system macro) 274
 special function registers (SFR)
 in Register window 151
 using as assembler symbols 73
 stack usage, computing 133
 Stack window 147
 stack.mac 233
 standard C, sizeof operator in C-SPY 74
 Start address (Edit Memory Access option) 168
 Start address (Fill option) 143
 Start address (Memory Save option) 141
 static analysis
 documentation for 22
 Statics window 87
 stdin and stdout, redirecting to C-SPY window 66
 Step Into (Debug menu) 47
 Step Into, description 55
 Step Out (Debug menu) 47
 Step Out, description 56
 Step Over (Debug menu) 47
 Step Over, description 55
 step points, definition of 54
 Stop address (Memory Save option) 141
 Stop Debugging (Debug menu) 46
 __strFind (C-SPY system macro) 274
 __subString (C-SPY system macro) 275
 Suppress download (debugger option) 309
 Symbolic Memory window 144
 Symbols window 92
 symbols, using in C-SPY expressions 72

T

target system, definition of 33
 __targetDebuggerVersion (C-SPY system macro) 275
 Terminal IO Log Files (Terminal IO Log Files option) 68
 Terminal I/O Log Files dialog box 67
 Terminal I/O window 59, 66
 terminology 23
 Text search (Find in Trace option) 190
 Time Axis Unit (Timeline window context menu) 184
 time interval, in Timeline window 198
 Timeline window 177
 Timeline (Simulator menu) 315
 --timeout (C-SPY command line option) 296
 timer interrupt, example 216
 Toggle Breakpoint (Code) (Call
 Stack window context menu) 65
 Toggle Breakpoint (Code) (Disassembly
 window context menu) 63
 Toggle Breakpoint (Log) (Call
 Stack window context menu) 66
 Toggle Breakpoint (Log) (Disassembly
 window context menu) 63
 Toggle Breakpoint (Trace Start) (Call
 Stack window context menu) 66
 Toggle Breakpoint (Trace Start) (Disassembly
 window context menu) 63
 Toggle Breakpoint (Trace Stop) (Call
 Stack window context menu) 66
 Toggle Breakpoint (Trace Stop) (Disassembly
 window context menu) 64
 Toggle source (Trace toolbar) 175
 __toLower (C-SPY system macro) 276
 tools icon, in this guide 24
 __toString (C-SPY system macro) 276
 __toUpper (C-SPY system macro) 277
 trace 171
 Trace Expressions window 188
 trace start and stop breakpoints, overview 108
 Trace Start breakpoints dialog box 186

Trace Stop breakpoints dialog box	187
Trace window	174
trace (calls), profiling source	194, 200
trace (flat), profiling source	194, 201
Trace (Simulator menu)	314
trace, in Timeline window	177
trademarks	2
Trigger (Forced Interrupt window context menu)	222
typographic conventions	23

U

UBROF	31
Unavailable, C-SPY message	75
Universal Binary Relocatable Object Format. <i>See</i> UBROF	
__unloadImage(C-SPY system macro)	277
Use command line options (debugger option)	312
Use Extra Images (debugger option)	310
Use flash loader (debugger option)	309
Use manual ranges (Memory Access Setup option)	158, 166
Use ranges based on (Memory Access Setup option)	157, 166
Used ranges (Memory Configuration option)	162
user application, definition of	33

V

Value (Fill option)	143
variables	
effects of optimizations	74
information, limitation on	74
using in C-SPY expressions	72
variance (interrupt property), definition of	213
Variance % (Edit Interrupt option)	221
Verify download (debugger option)	309
version number	
of this guide	2
Viewing Range dialog box	185
Viewing Range (Sampled Graphs window context menu)	105
Viewing Range (Timeline window context menu)	183

visualSTATE, C-SPY plugin module for	34
--------------------------------------	----

W

warnings icon, in this guide	24
Watch window	83
using	71
web sites, recommended	23
while (macro statement)	243
windows, specific to C-SPY	48
With I/O emulation modules (linker option), using	66
__writeFile (C-SPY system macro)	278
__writeFileByte (C-SPY system macro)	278
__writeMemoryByte (C-SPY system macro)	279
__writeMemory8 (C-SPY system macro)	279
__writeMemory16 (C-SPY system macro)	279
__writeMemory32 (C-SPY system macro)	280

Z

zone	
defined in device description file	132
in C-SPY	132
part of an absolute address	127
Zone (Edit SFR option)	156
Zoom (Sampled Graphs window context menu)	104
Zoom (Timeline window context menu)	183

Symbols

__cancelAllInterrupts (C-SPY system macro)	250
__cancelInterrupt (C-SPY system macro)	251
__clearBreak (C-SPY system macro)	251
__closeFile (C-SPY system macro)	251
__delay (C-SPY system macro)	252
__disableInterrupts (C-SPY system macro)	252
__driverType (C-SPY system macro)	252
__enableInterrupts (C-SPY system macro)	253
__evaluate (C-SPY system macro)	253

__fillMemory8 (C-SPY system macro)	254
__fillMemory16 (C-SPY system macro)	255
__fillMemory32 (C-SPY system macro)	256
__fmessage (C-SPY macro statement)	243
__isBatchMode (C-SPY system macro)	257
__loadImage (C-SPY system macro)	257
__memoryRestore (C-SPY system macro)	258
__memorySave (C-SPY system macro)	259
__message (C-SPY macro statement)	243
__messageBoxYesCancel (C-SPY system macro)	260
__messageBoxYesNo (C-SPY system macro)	260
__openFile (C-SPY system macro)	261
__orderInterrupt (C-SPY system macro)	262
__popSimulatorInterruptExecutingStack (C-SPY system macro)	263
__readFile (C-SPY system macro)	263
__readFileByte (C-SPY system macro)	264
__readMemoryByte (C-SPY system macro)	264
__readMemory8 (C-SPY system macro)	264
__readMemory16 (C-SPY system macro)	265
__readMemory32 (C-SPY system macro)	265
__registerMacroFile (C-SPY system macro)	266
__resetFile (C-SPY system macro)	266
__setCodeBreak (C-SPY system macro)	267
__setDataBreak (C-SPY system macro)	268
__setDataLogBreak (C-SPY system macro)	269
__setLogBreak (C-SPY system macro)	270
__setSimBreak (C-SPY system macro)	271
__setTraceStartBreak (C-SPY system macro)	272
__setTraceStopBreak (C-SPY system macro)	273
__smessage (C-SPY macro statement)	243
__sourcePosition (C-SPY system macro)	274
__strFind (C-SPY system macro)	274
__subString (C-SPY system macro)	275
__targetDebuggerVersion (C-SPY system macro)	275
__toLower (C-SPY system macro)	276
__toString (C-SPY system macro)	276
__toUpper (C-SPY system macro)	277
__unloadImage (C-SPY system macro)	277
__writeFile (C-SPY system macro)	278

__writeFileByte (C-SPY system macro)	278
__writeMemoryByte (C-SPY system macro)	279
__writeMemory8 (C-SPY system macro)	279
__writeMemory16 (C-SPY system macro)	279
__writeMemory32 (C-SPY system macro)	280
-f (cspybat option)	293
-p (C-SPY command line option)	295
--backend (C-SPY command line option)	290
--code_coverage_file (C-SPY command line option)	291
--cycles (C-SPY command line option)	291
--debugfile (cspybat option)	292
--disable_interrupts (C-SPY command line option)	292
--download_only (C-SPY command line option)	292
--flash_loader (C-SPY command line option)	293
--macro (C-SPY command line option)	294
--macro_param (C-SPY command line option)	294
--mapu (C-SPY command line option)	295
--plugin (C-SPY command line option)	296
--silent (C-SPY command line option)	296
--timeout (C-SPY command line option)	296

Numerics

1x Units (Memory window context menu)	149
1x Units (Symbolic Memory window context menu)	146
2x Units (Memory window context menu)	149
4x Units (Memory window context menu)	149
8x Units (Memory window context menu)	139